# Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation

Pierre-Charles David

**UNIVERSITÉ DE NANTES**
**UFR SCIENCES ET TECHNIQUES**

**École Doctorale**

**Sciences et Technologies de l'Information et des Matériaux**

2005

**Thèse de DOCTORAT**

Spécialité : INFORMATIQUE

*Présentée et soutenue publiquement par*

**Pierre-Charles DAVID**

*le 1er juillet 2005*

*à l'École des Mines de Nantes*

# Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation

Jury

| | | | |
|---|---|---|---|
| Président | : | Laurence DUCHIEN | LIFL / Université de Lille |
| Rapporteurs | : | Daniel HAGIMONT, Chargé de recherche | INRIA |
| | | Michel RIVEILL, Professeur | ESSI / Université de Nice |
| Examinateurs | : | Laurence DUCHIEN, Professeur | LIFL / Université de Lille |
| | | Frédéric BENHAMOU, Professeur | Université de Nantes |
| | | Pierre COINTE, Professeur | École des Mines de Nantes |
| | | Thomas LEDOUX, Maître-assistant | École des Mines de Nantes |
| | | Thierry COUPAYE, Chercheur | France Télécom division R&D, *membre invité* |

**Directeur de thèse : Pierre Cointe**
Laboratoire : LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE (CNRS FRE 2729).
Université de Nantes
2, rue de la Houssinière, F-44322 NANTES CEDEX 3
École des Mines de Nantes
4, rue Alfred Kastler, F-44307 NANTES CEDEX 3

N° ED 366-204

# DÉVELOPPEMENT DE COMPOSANTS FRACTAL ADAPTATIFS : UN LANGAGE DÉDIÉ À L'ASPECT D'ADAPTATION

**Pierre-Charles David**

*favet neptunus eunti*

**Université de Nantes**

Pierre-Charles DAVID

***Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation***

# Remerciements

Je remercie tout d'abord Pierre Cointe, qui a dirigé cette thèse, et Thomas Ledoux qui m'a encadré pendant presque quatre ans.

Je remercie Daniel Hagimont et Michel Riveill d'avoir été rapporteurs de cette thèse, ainsi que les autres membres du jury : Laurence Duchien, Frédéric Benhamou et Thierry Coupaye.

Merci à Rémi Douence, Éric Tanter, Yann-Gaël Guéhéneuc, Andrès Farias, Marc Ségura-Devillechaise et Gustavo Bobeff pour leur amitié.

Je tiens aussi à remercier les Drs Martine Couet et Michel Meigner sans l'aide desquels je n'aurais sans doute pas pu aller jusqu'au bout.

Enfin, je remercie mes parents, qui m'ont toujours laissé libre de mes choix et soutenu quels qu'ils soient.

## Résumé

Les contextes toujours plus variés et dynamiques dans lesquels les logiciels actuels s'exécutent leurs imposent de s'adapter de façon *autonome* à ces changements. L'objectif de cette thèse est de faciliter le développement de telles *applications adaptatives*, en considérant l'adaptation comme un *aspect* qui doit être développé séparément du reste de l'application afin de pouvoir y être intégré et modifié dynamiquement. Pour cela nous proposons SAFRAN, une extension du modèle de composants Fractal permettant d'associer dynamiquement des politiques d'adaptation aux composants d'une application. Ces politiques sont programmées dans un langage dédié sous la forme de règles réactives. Leur exécution repose d'une part sur WildCAT, un système permettant de détecter les évolutions du contexte d'exécution (quand adapter ?), et d'autre part sur FScript, un langage dédié pour la reconfiguration dynamique consistante de composants Fractal (comment adapter ?).



**Mots-clés :** logiciels adaptatifs, séparation des préoccupations, politiques d'adaptation, langages dédiés, règles réactives, reconfiguration dynamique, sensibilité au contexte, composants logiciels, Fractal



## Abstract

The increasingly diverse and dynamic contexts in which current applications are run imposes them to adapt and to become more *autonomous*. The goal of this thesis is to ease the development of such *self-adaptive applications*, by considering adaptation as an *aspect* which should be defined separately from the rest of the application, so as to be integrated and modified dynamically. To this end we propose SAFRAN, an extension of the Fractal component model enabling dynamic association of adaptation policies to the components of an application. These policies are programed using a Domain-Specific Language in the form of reactive rules. Their execution harnesses WildCAT, a context-awareness system which can detect changes in the execution context (when to adapt?), and FScript, a language dedicated to dynamic and consistent reconfigurations of Fractal components (how to adapt?).



**Keywords:** self-adaptive software, separation of concerns, adaptation policies, domain-specific languages, reactive rules, dynamic reconfiguration, context-awareness, software components, Fractal

*Pour moi, je nageais à l'aventure, poussé à la fois par le vent et la marée. J'essayais parfois, mais en vain, de toucher le fond ; finalement, alors que j'étais sur le point de m'évanouir, et dans l'impossibilité de prolonger la lutte, je m'aperçus que j'avais pied.*

— Les Voyages de Gulliver, 1726 (Jonathan Swift)

# Table des matières

# Introduction

## Motivation et problématique

Avec la prolifération des plates-formes d'exécution aux caractéristiques techniques parfois très différentes et la démocratisation des réseaux, notamment sans fil, nous sommes confrontés à des contextes d'exécution de plus en plus variables :

- variations dans *l'espace*, avec une grande diversité de plate-formes couvrant un large spectre en terme de ressources disponibles (des systèmes embarqués aux grilles de calculs), ces machines hétérogènes étant de plus en plus reliées par le réseau et donc interdépendantes ;
- variations dans *le temps*, le contexte d'exécution d'une application donnée évoluant de plus en plus pendant son exécution : disponibilité des ressources matérielles et logicielles, mobilité (nomadisme), etc. ;
- enfin, variations *des utilisations et des utilisateurs*, une même application pouvant être utilisée dans des situations qui nécessitent des modes de fonctionnement différents (par exemple un téléphone portable au bureau, en voiture, dans un lieu public...), et par des utilisateurs aux niveaux d'expertise et aux besoins spécifiques.

Cette situation rend le développement d'applications de plus en plus complexe, car il est souvent difficile de connaître au moment du développement les conditions précises dans lesquelles les applications seront utilisées. Même lorsque cela est possible, ces conditions seront de toute façon amenées à évoluer de façon imprévisible au cours de la vie de l'application. L'approche traditionnelle qui consiste à isoler les applications de leur contexte en introduisant des niveaux d'abstraction (système d'exploitation, machines virtuelles, intergiciels...) ne fonctionne que si les variations ne sont pas trop importantes et surtout suppose que la gestion des variations contextuelles peut se faire indépendemment de la sémantiques des applications, ce qui n'est pas toujours le cas.

Plutôt que d'ignorer le contexte d'exécution, nous pensons que les applications doivent au contraire en être conscientes (*context-aware*) afin de pouvoir s'y adapter. L'approche que nous proposons consiste donc à créer des *applications adaptatives*, capables de s'adapter elles-mêmes de façon la plus autonome possible aux différents types de variations qu'elles rencontrent. Lorsque son contexte d'exécution évolue, le rôle d'une adaptation est alors non seulement de permettre à l'application de continuer à fonctionner, mais aussi de tirer le meilleur parti possible des nouvelles possibilités qui peuvent apparaître.

Le besoin de construire des applications qui s'adaptent à leur environnement n'est pas nouveau. Cependant, toutes les évolutions récentes citées plus haut rendent cet objectif à la fois plus important et plus difficile à atteindre que jamais. Les techniques *ad hoc* utilisées généralement, dans lesquelles les décisions d'adaptation sont « câblées » dans l'application, ne sont pas suffisantes. Intégrer directement dans l'application le code nécessaire pour détecter les évolutions du contexte et y réagir est incompatible avec la situation actuelle où il est le plus souvent impossible de prévoir les circonstances dans lesquelles une application sera utilisée, et encore moins les réactions appropriées. De plus, une telle approche « pollue » le code métier de l'application par des préoccupations non-fonctionnelles comme l'observation de l'environnement et le choix de la réaction appropriée.

On peut donc résumer la problématique du développement d'applications adaptatives par les points suivants :

1. En terme de méthodologie de développement, il est impossible de prévoir toutes les circonstances dans lesquelles l'application sera utilisée, et encore moins les réactions appropriées à chaque cas.
2. D'un point de vue plus technique, le mélange du code métier et du code chargé d'adapter celui-ci implique une complexité accrue de l'application, qui rend son développement et sa maintenance difficile, et diminue la réutilisabilité des composants logiciels en ne les rendant utilisable que dans les circonstances prévues explicitement.

L'objectif de cette thèse est de *faciliter le développement d'applications adaptatives* en proposant une approche qui prend en compte ces deux problèmes. Afin de remplir cet objectif, la thèse défendue dans ce document peut se résumer en trois points :

1. Grâce à la possibilité de reconfigurer dynamiquement leur architecture – y compris de façon non anticipée –, les applications à base de *composants* forment un bon substrat pour la création d'applications adaptatives.
2. Considérer l'adaptation comme un *aspect* et permettre son développement séparément du code métier – aussi bien sur le plan spatial que sur le plan temporel – offre de nombreux avantages (modularité, dynamicité, réutilisation) par rapport aux approches classiques *ad hoc*.
3. Plus spécifiquement, le processus d'adaptation étant par essence réactif, le paradigme des *règles réactives* ECA (*Event–Condition–Action*) est bien adapté à la définition de l'aspect d'adaptation. On parlera alors de *politiques d'adaptations*.

# Contributions

Pour prouver cette thèse, nous avons créé le système SAFRAN (*Self-Adaptive Fractal CompoNents*), qui constitue notre principale contribution. SAFRAN est une extension du modèle de composants Fractal [Bruneton et al., 2003] pour la création d'applications adaptatives. Sa conception est basée sur les trois principes énoncés ci-dessus : *(i)* l'utilisation du modèle de composants Fractal pour la création d'applications adaptables par reconfiguration dynamique, *(ii)* la modularisation des *politiques d'adaptation* d'une application sous forme d'*aspects*, développées séparément du code des composants auxquels elles peuvent ensuite être attachées et détachées dynamiquement, et *(iii)* l'utilisation d'un *langage dédié* basé sur la notion de *règle réactive* pour programmer ces politiques d'adaptation.

Les contributions de cette thèse sont les suivantes :

1. Une analyse de la notion d'adaptation appliquée au logiciel débouchant sur une liste des caractéristiques essentielles des logiciels adaptatifs (Chapitre 1).
2. Deux extensions du modèle de composants Fractal destinées à augmenter ses capacités d'adaptation. La première permet des adaptations non-anticipées grâce à la réflexion comportementale (Section 4.3.1) et la seconde ajoute le support des contraintes architecturales métier qui permet de garantir l'intégrité structurelle de l'application malgré les reconfigurations (Section 4.3.2).
3. WildCAT, un *framework* générique pour l'observation et le raisonnement sur le contexte d'exécution des applications (Chapitre 5). WildCAT permet de rendre une application consciente de son contexte (*context-aware*), condition *sine qua non* pour pouvoir s'y adapter.
4. FScript, un langage dédié pour la spécification et l'exécution de reconfigurations dynamiques consistantes de composants Fractal (Chapitre 6). Par sa conception et son implémentation, FScript offre des garanties sur la consistance des reconfigurations appliquées aux composants, évitant ainsi qu'une adaptation puisse rendre l'application inutilisable.
5. Enfin, le système SAFRAN lui-même, qui regroupe tous ces éléments grâce à un langage dédié basé sur la notion de règles actives : WildCAT permet de détecter les modifications du contexte d'exécution qui doivent déclencher une adaptation, et FScript d'exprimer les reconfigurations à appliquer. Ce langage permettant de développer facilement des politiques d'adaptation de façon modulaire. Une extension de Fractal permet ensuite d'attacher dynamiquement ces politiques aux composants Fractal d'une application, faisant de ces derniers des *composants adaptatifs* (Chapitres 4 et 7).

Par rapport aux deux problèmes identifiés plus haut concernant le développement d'applications adaptatives, l'approche utilisée par SAFRAN, basée sur la notion d'*aspect d'adaptation*, a les avantages suivants :

- La modularisation (découplage spatial) de la politique d'adaptation évite de polluer le code purement métier d'une application par la logique d'adaptation.
- La possibilité d'attacher et de détacher dynamiquement des politiques d'adaptation aux composants d'une application (découplage temporel) évite d'avoir à prévoir à l'avance tous les cas de figure possibles et les adaptations appropriées à chacun.

## Plan du document

Ce document est constitué de deux parties, la première étudiant la problématique et les solutions existantes, et la seconde décrivant nos différentes contributions.

Le chapitre 1 définit plus précisément notre problématique, analyse la notion d'adaptation appliquée au domaine du logiciel, et identifie les critères qui nous serviront à évaluer les solutions proposées, la nôtre comme celles des autres. Les deux chapitres suivants étudient l'existant : le chapitre 2 introduit différents concepts généraux sur lesquels notre approche est fondée (séparation des préoccupations, réflexion, composants logiciels et langages dédiés), alors que le suivant (Chapitre 3) étudie un certain nombre de propositions existantes pour le développement d'applications adaptatives, en regard des critères identifiés au préalable. Ce chapitre conclut la première partie en identifiant les différentes stratégies habituellement utilisées et en choisissant celles qui nous semblent les plus appropriées.

La seconde partie débute par le chapitre 4 qui présente les grandes lignes de l'architecture de SAFRAN et présente rapidement les différents éléments qui constituent ce système. Les chapitres suivants détaillent chacun de ces éléments : tout d'abord WildCAT (Chapitre 5), un système permettant de rendre une application sensible à son contexte d'exécution, puis FScript (Chapitre 6), un langage dédié pour programmer des reconfigurations dynamiques consistantes de composants Fractal, et pour finir (Chapitre 7) le langage dédié SAFRAN lui-même, qui repose sur WildCAT et FScript et permet d'écrire des politiques d'adaptation à base de règles réactives, puis de les attacher dynamiquement aux composants Fractal d'une application. Le chapitre 8 valide ensuite notre proposition en illustrant l'utilisation de SAFRAN sur plusieurs applications exemples.

Enfin, la conclusion fait la synthèse de nos contributions et des résultats obtenus, et identifie un certain nombre de pistes pour de travaux futurs en vue d'améliorer SAFRAN.

# Première partie

# Problématique et état de l'art

# Chapitre 1

# Définition et analyse préliminaire de la problématique

**Sommaire**



L'OBJECTIF de ce chapitre est double : tout d'abord, définir précisément la problématique de nos travaux et en circonscrire la portée ; ensuite, proposer une analyse de la notion d'adaptation en général et de son application au domaine du logiciel en particulier, et inventorier les critères qui nous serviront à évaluer les solutions proposées, la nôtre comme celles des autres.

## 1.1 Définition de la problématique et des objectifs

Cette section détaille la problématique de nos travaux de façon plus précise que dans l'introduction, et dégage de cette définition les *objectifs* à atteindre et les *critères* qui nous ont servi d'une part à évaluer les solutions existantes dans le chapitre 3, et d'autre part à guider la conception de notre proposition.

### 1.1.1 Constats et motivation

Avec la prolifération des plates-formes d'exécution aux caractéristiques techniques parfois très différentes – du téléphone portable aux grilles de calcul – et la démocratisation des réseaux, notamment sans fil, dans lesquels la disponibilité des ressources peut varier de façon imprévisible en cours d'exécution (processeur, mémoire, ressources logicielles, ...), nous sommes plus que jamais confrontés à l'*hétérogénéité* et la *dynamicité* des environnements dans lesquels nos applications doivent fonctionner. Ces deux types de *variabilité*, spatiale et temporelle, rendent le développement de nouvelles applications de plus en plus complexe, car il est souvent difficile, voire impossible, de connaître au moment du développement les conditions précises dans lesquelles les applications seront utilisées. Une troisième source de difficulté souvent ignorée est la *diversité des utilisateurs* et des utilisations, qui implique que les objectifs et les priorités d'une application peuvent eux aussi varier selon les besoins.

**Diversité des plates-formes (variabilité spatiale)**

Le problème de l'hétérogénéité est accentué aujourd'hui par la prolifération des plates-formes d'exécution couvrant un large spectre de caractéristiques techniques, de la carte à puce programmable aux grilles de machines multi-processeurs, en passant par les téléphones portables, assistants personnels (PDA[1]) et ordinateurs plus classiques. De nouvelles plates-formes apparaissent régulièrement, chacune avec une architecture spécifique : nouveaux microprocesseurs, systèmes d'exploitation, périphériques physiques...

**Ressources limitées.** La nature même de beaucoup de ces nouvelles plates-formes augmente la difficulté du développement d'applications les ciblant. La plupart d'entre elles ont des capacités limitées par rapport à un ordinateur de bureau, en terme de capacité de traitement, de stockage et d'affichage en particulier. Étant données ces limitations, ces plates-formes dépendent fortement de leurs capacités à communiquer avec d'autres systèmes plus puissants. Par exemple, un assistant personnel n'a souvent qu'une capacité de stockage locale relativement réduite ; son utilisation requiert de fréquentes synchronisations avec un ordinateur de bureau. Les applications conçues pour une telle plate-forme doivent avoir conscience de cette limitation car cela signifie qu'une partie de leurs données peut ne pas être accessible à tout moment. Elles doivent donc être capables de manipuler de façon cohérente des données partielles, comme par exemple des courriers électroniques dont les pièces jointes ont été supprimées, et de fusionner celles-ci avec les données complètes lors d'une synchronisation.

**Hétérogénéité.** Au-delà des limitations de certaines plates-formes, la diversité même pose problème. Il devient de moins en moins intéressant de développer une application spécifiquement pour une plate-forme ou une classe de plate-forme. La plupart des utilisateurs disposent de plusieurs ordinateurs de capacités différentes – par exemple un ordinateur de bureau, un assistant personnel ou téléphone portable, et parfois un ordinateur portable – et souhaitent naturellement pouvoir utiliser leurs applications sur tous ces systèmes sans limitations arbitraires. Si les aspects techniques de la portabilité de code ont été résolus depuis longtemps avec par exemple des machines virtuelles comme la JVM[2] ou le CLR[3], cela est loin d'être suffisant. Lorsqu'elle s'exécute sur un téléphone portable, une application *ne doit pas* se comporter de la même manière que sur un ordinateur de bureau, même si la portabilité de son code rend la chose techniquement faisable. Les contraintes de traitement, d'affichage et d'autonomie – entre autres – y sont beaucoup plus fortes. Une telle application devra être capable d'*adapter son comportement* aux caractéristiques de son hôte, par exemple en réduisant ses fonctionnalités ou en choisissant des algorithmes moins gourmands en ressources, quitte à obtenir des résultats approximatifs (si cela est pertinent). La même application tournant sur un ordinateur de bureau moderne devra au contraire être capable de tirer partie des spécificités de cette plate-forme. Il ne s'agit pas d'être « simplement » capable de fonctionner sur des systèmes aux ressources limitées, mais de tirer partie au mieux de caractéristiques – limitations ou capacités spécifiques – de son hôte.

**Distribution.** De plus en plus, une application ne s'exécute plus sur *un* hôte spécifique, mais est distribuée sur un ensemble de machines. La façon dont les différentes parties d'une application distribuée est répartie sur ses différents hôtes, qui peuvent avoir des caractéristiques différentes, peut avoir un impact très important sur ses performances. L'utilisation de plus en plus répandue de grilles de calcul pour résoudre des problèmes très complexes ou gourmands en mémoire est un exemple significatif de cette tendance. La distribution des traitements permet d'utiliser de façon plus efficace les parcs de machines existantes. Cependant, pour tirer réellement partie de cette approche de nombreux problèmes doivent être résolus : répartition de la charge dynamique entre les noeuds, utilisation optimale des ressources spécifiques de chaque noeud, sécurité, administration la plus automatisée possible, etc. Les applications distribuées ne sont plus limitées aux gros systèmes. On retrouve ces problématiques à tous les niveaux, y compris à celui des assistants personnels ou des téléphones portables. Comme on l'a déjà vu plus haut, les

[1] Portable Digital Assistant
[2] Java Virtual Machine
[3] Common Language Runtime

ressources limitées de ces plates-formes les rendent très dépendants de leurs moyens de communication avec des systèmes plus puissants. Étant donnés les capacités de communication toujours plus sophistiquées disponibles, il est de plus en plus intéressant d'être capable d'en tirer partie pour par exemple déléguer certaines tâches complexes à des systèmes puissant et n'effectuer localement que l'affichage des résultats et l'interaction avec l'utilisateur.

### Des systèmes de plus en plus dynamiques (variabilité temporelle)

La section précédente décrit un certain nombre de problèmes dûs à la *diversité* des plates-formes d'exécution. Cependant, cette hétérogénéité n'est qu'un aspect du problème.

**Disponibilité des ressources.** En effet, même si l'on connaît à l'avance le type de plate-forme sur laquelle une application sera déployée, le contexte dynamique de l'application, et en particulier la disponibilité des ressources (processeur, mémoire, réseau...), peut varier énormément. Or, tous ces paramètres peuvent avoir un impact important sur le fonctionnement de l'application, en particulier au niveau des performances ou plus généralement de la qualité de service (*QoS*). Par exemple, dans le cas d'une application de diffusion de vidéo par le réseau, le choix de l'algorithme de codage et de décodage des données se fait en général en fonction des caractéristiques de la liaison réseau (débit, temps de latence...), mais ces caractéristiques peuvent se dégrader de façon imprévisible pendant l'exécution, entraînant une perte de la qualité perçue par l'utilisateur (ralentissements, désynchronisation entre le son et l'image, etc.). L'application devrait être capable de réagir à ces évolutions pour s'adapter aux nouvelles conditions, par exemple en demandant au serveur d'augmenter le taux de compression pour diminuer la bande passante utilisée. De plus, les normes telles USB[4] ou IEEE[5] 1394 (FireWire) permettent aux utilisateurs d'ajouter ou de retirer des périphériques « à chaud », c'est-à-dire pendant que les applications s'exécutent. Si les systèmes d'exploitation actuels sont capables de réagir à ces événements, ce n'est pas le cas de la plupart des applications.

**Nomadisme.** Un autre facteur important qui augmente la complexité des systèmes informatiques est la mobilité croissante des utilisateurs et des ordinateurs. Une application ne peut plus compter sur un environnement figé une fois pour toute (ou évoluant très rarement) mais doit au contraire s'attendre à ce que l'infrastructure dans laquelle elle s'exécute – en particulier l'infrastructure réseau – évolue constamment en fonction des déplacements de l'utilisateur. Pour tenter de résoudre ce problème, de nombreux travaux existent actuellement ayant pour objectif la création de *réseaux ad hoc* [Corson et al., 1999; Frodigh et al., 2000], c'est-à-dire des réseaux sans fils connectant spontanément des hôtes physiquement proches, et dont la topologie évolue automatiquement en fonction de l'arrivée ou du départ de ses membres. Les évolutions d'un tel réseau étant totalement imprévisibles, les applications doivent être préparées à y réagir automatiquement, par exemple en échangeant automatiquement des données (carte de visite...) avec un nouvel hôte, en notifiant l'utilisateur de l'arrivée d'un hôte spécifique ou en lançant une sauvegarde de ces données si l'ordinateur de bureau de son utilisateur est tout-à-coup accessible. Au delà des nouvelles opportunités d'interaction offertes par ces réseaux, de nombreux problèmes se posent, concernant par exemple la sécurité ou la vie privée des utilisateurs. Étant données les spécificités d'un tel environnement, les applications ne peuvent pas le traiter comme un réseau « normal » simplement plus dynamique ; elles doivent s'adapter à ces particularités et en tirer partie le plus possible.

### Diversité des utilisateurs et des utilisations (variabilité des besoins)

La démocratisation de l'outil informatique et l'apparition de nouveaux types d'ordinateurs spécialisés et plus simples à manipuler a introduit de nouveaux profils d'utilisateurs. Si autrefois la plupart des utilisateurs étaient soit des spécialistes soit des utilisateurs professionnels formés à l'utilisation des quelques applications dont ils avaient besoin, aujourd'hui tout le monde ou presque a accès à ces technologies.

[4]Universal Serial Bus

[5]Institute of Electrical and Electronics Engineers

Cependant, tout le monde n'est pas devenu un expert pour autant ; il existe une grande diversité dans les niveaux d'expertise des utilisateurs potentiels d'un même système. Certains peuvent avoir une bonne connaissance générale de l'utilisation d'un ordinateur (bien qu'en général limitée à un système d'exploitation spécifique), mais ne pas connaître une application particulière ou un domaine spécifique. À l'inverse, certaines personnes sont ultra-spécialisées, et maîtrisent parfaitement un ou deux logiciels particuliers, par exemple dans le cadre professionnel, mais ne sont pas capables de réaliser des tâches relativement simples en dehors de ce cadre. L'interface et le niveau de contrôle qu'une application offre à ses utilisateurs est en général figée, et cible une classe d'utilisateurs particuliers. Or, il serait très intéressant pour une application – et pour ses utilisateurs – d'être utilisable efficacement aussi bien par des débutants que par des utilisateurs avancés. Ce n'est cependant pas trivial, car cela implique d'adapter l'interface utilisateur, voire le mode de fonctionnement de l'application (par exemple par l'utilisation d'assistants pour guider les débutants) aux compétences de l'utilisateur, qui d'ailleurs évolueront au fur et à mesure qu'il se familiarisera avec l'application.

Le niveau d'expertise n'est pas la seule chose qui différencie les utilisateurs. Certains utilisateurs peuvent avoir des handicaps physiques ou mentaux plus ou moins importants, qui les empêchent d'utiliser la plupart des applications normalement. Il existe des technologies permettant de pallier la plupart des handicaps, comme par exemple la synthèse ou la reconnaissance vocale, l'utilisation de « loupes » pour agrandir certaines parties de l'écran, l'utilisation de thèmes de couleurs très contrastés, ou bien encore de nombreux périphériques d'entrée ou de sortie spécialisés (terminaux braille, etc.). Peu d'applications grand public sont aujourd'hui capables de tirer partie de ces technologies pour s'adapter aux handicaps de leurs utilisateurs. Cela est sans doute dû en partie au nombre faible de personnes concernées – relativement à la population globale – mais aussi à la difficulté technique de développer une application suffisamment souple et « intelligente » pour s'adapter à ses utilisateurs.

Enfin, les technologies récentes qui favorisent le nomadisme permettent d'utiliser les applications presque en toutes circonstances. Or, le comportement approprié d'une application peut être différent selon les situations dans lesquelles son utilisateur se trouve :

- le contexte « social » : un lecteur multimédia par exemple, peut décider de remplacer la bande son d'un film par des sous-titres si l'utilisateur se trouve dans un lieu public (transport en commun), ou un économiseur d'écran peut se désactiver si l'utilisateur utilise son ordinateur pour une présentation.
- l'environnement physique : le même lecteur multimédia peut vouloir adapter le volume sonore au bruit ambiant, pour permettre à l'utilisateur de distinguer les dialogues, ou bien un assistant personnel peut ajuster le contraste de son écran à la luminosité extérieure (si ses batteries le permettent).
- l'activité de l'utilisateur : ne pas le déranger avec des notifications de mail s'il est concentré sur la rédaction de sa thèse, ou activer le répondeur d'un téléphone pendant qu'il est en réunion pour ne pas le déranger avec la sonnerie.
- des préférences utilisateur exprimées explicitement, comme par exemple le choix de donner la priorité à la bande son lors d'une vidéo-conférence si la bande passante disponible ne permet pas d'avoir à la fois le son et l'image de bonne qualité.

### Nécessité et difficulté de construire des applications adaptatives

Dans le contexte décrit ci-dessus, les nouvelles applications doivent non seulement pouvoir *être adaptées*, mais aussi être capable de *s'adapter* elle-mêmes de façon *autonome* [Kephart, 2002] aux nombreux et divers contextes d'exécution auxquels elles seront confrontées, et aux évolutions – de plus en plus dynamiques – de ceux-ci. Ces adaptations doivent permettre à l'application de continuer à fonctionner correctement malgré ces évolutions, et de tirer parti au mieux des possibilités spécifiques apparues en cours d'exécution, comme par exemple lors de l'ajout d'un nouveau périphérique. Il est important que ces systèmes soient le plus autonomes possible. En effet, qu'il s'agisse de systèmes d'informations d'entreprise ou d'applications destinées à l'utilisateur final, la complexité des logiciels et les temps de réaction nécessaires aux adaptations rendent impossible la sollicitation d'une intervention humaine.

Historiquement, la tendance a toujours été d'isoler le plus possible les applications de leur environ-

nement pour les rendre indépendantes de facteurs externes qui ont tendance à évoluer rapidement et de façon imprévisible. La notion d'intergiciel (*middleware*) est particulièrement représentative de cette tendance. On se rend compte aujourd'hui que cette approche, en privilégiant le plus petit dénominateur commun à toutes les plates-formes conduit à la sous-utilisation des ressources spécifiques disponibles. En conséquence, la nouvelle tendance est à la construction d'applications *adaptatives*, « conscientes » (*aware*) de l'environnement dans lequel elles s'exécutent et capables de tirer parti de ses spécificités.

Cependant, atteindre ce niveau d'adaptation et d'autonomie est loin d'être évident. Pour commencer, la nature dynamique de la plupart des évolutions du contexte d'exécution les rendent difficilement prévisibles, et encore moins contrôlables.

Les solutions trop figées et fermées ne sont pas satisfaisantes :

- soit une telle application est spécialisée pour être utilisée dans un contexte très spécifique, mais elle ne touche alors qu'un « marché » restreint et ne prend pas en compte les évolutions possibles de ce contexte ;
- soit elle est généraliste, conçue pour fonctionner dans n'importe quel contexte, mais elle n'est alors pas capable de tirer partie des spécificités de son environnement et doit se contenter du plus petit sous-ensemble de fonctionnalités standards, et est encore une fois incapable de gérer les évolutions de son contexte d'exécution.

Pour pouvoir être réellement adaptatives, ces applications doivent donc être – au moins en partie – ouvertes et extensibles, afin de pouvoir évoluer en même temps que leur environnement. Il n'est cependant pas évident de trouver le bon équilibre entre forme et ouverture [Cointe et al., 2004], afin de garantir à la fois l'évolutivité et l'adaptabilité de l'application, et le maintient de sa cohérence interne.

Il n'y a pas actuellement de méthodologie permettant de créer ce genre d'applications adaptatives facilement. Les programmeurs font donc appel à des techniques *ad hoc*, forcément limitées et non réutilisables. Sur le plan purement technologique la création d'applications complètement ouvertes ne pose aucun problème : les langages et environnements d'exécution modernes comme Java et .NET supportent par exemple très bien le chargement dynamique de code et la réflexion. Le problème se situe plutôt en terme de méthodologie de développement : quelles technologies utiliser, et comment les intégrer dans le développement des applications pour en tirer le maximum de bénéfices en terme d'adaptabilité tout en en minimisant les effets négatifs.

### 1.1.2 Problématique de la thèse

Le besoin de construire des applications qui s'adaptent à leur environnement n'est pas nouveau. Cependant, toutes les évolutions récentes décrites ci-dessus rendent cet objectif à la fois de plus en plus important et de plus en plus difficile à atteindre.

Jusqu'ici, le développement d'applications adaptatives se faisait généralement de façon *ad hoc*, en tentant d'anticiper au moment du développement les futures conditions d'exécution de l'application, et en intégrant directement dans celle-ci le code nécessaire pour détecter ces évolutions et y réagir. La situation actuelle rend cette approche caduque : il est impossible de prévoir à l'avance les circonstances dans lesquelles une application sera utilisée et encore moins de prévoir les réactions appropriées, non seulement à cause de leur multiplicité mais aussi car la « bonne » réaction peut dépendre de l'utilisateur (variabilité des besoins).

Au-delà du problème consistant à savoir quand et comment adapter une application, les techniques utilisées pour mettre en œuvre ces adaptations posent plusieurs problèmes en terme de génie logiciel. Tout d'abord, l'intégration du code chargé de l'adaptation dans l'application même augmente sa complexité : le code métier se retrouve « pollué » par des préoccupations non-fonctionnelles comme l'observation de l'environnement pour détecter ses évolutions et le choix de la réaction appropriée. De plus, les composants (au sens large) développés de cette manière sont faiblement réutilisables : ils ne sont capables de fonctionner que dans les situations anticipées pendant leur développement, voire uniquement dans une application spécifique. En effet, dès qu'une application atteint une certaine taille, les adaptations de ces différentes parties doivent être coordonnées pour rester globalement cohérentes, ce qui augmente le couplage entre ces différentes parties, par rapport à ce que le code métier seul nécessite.

On peut donc résumer la problématique du développement d'applications adaptatives par les points suivants :

1. En terme de méthodologie de développement, il est impossible de prévoir toutes les circonstances dans lesquelles l'application sera utilisée, et encore moins les réactions appropriées à chaque cas.
2. D'un point de vue plus technique, le mélange du code métier et du code chargé d'adapter celui-ci implique une complexité accrue de l'application, qui rend son développement et sa maintenance difficile, et diminue la réutilisabilité des composants en ne les rendant utilisable que dans les circonstances prévues explicitement.

L'objectif de cette thèse est de *faciliter le développement d'applications adaptatives en proposant une nouvelle approche* prenant en compte les problèmes que l'on vient de citer.

Le premier point ci-dessus implique qu'une application adaptative doit être *ouverte*, c'est-à-dire capable d'évoluer ou d'être étendue de façons non prévues explicitement par ses développeurs. Cependant, rendre une application trop flexible peut être contre-productif ; une application donnée est conçue comme une solution à un problème spécifique, et sa conception initiale reflète cette solution. Permettre des modifications trop importantes risque de nuire à l'efficacité de l'application en altérant sa nature même. Découvrir le bon compromis entre forme et ouverture, ou entre structure et flexibilité, est donc primordial. Dans notre cas particulier, nous devons permettre de construire des applications suffisamment souples pour être adaptées facilement à de nombreuses situations, tout en conservant leur intégrité.

Le second point, concernant plutôt les techniques de génie logiciel employées, requiert un couplage le plus lâche et le plus dynamique possible entre deux « types » de code : d'une part le code métier de l'application et d'autre part celui chargé d'adapter celui-ci (ce qui, comme nous le verrons tout au long de ce document, implique un certain nombre de tâches différentes et complexes). Plus ce couplage sera lâche, plus il sera possible de développer le code métier de façon abstraite et générale, sans avoir à se préoccuper des problèmes d'adaptation, qui seront résolus séparément. Plus le couplage sera dynamique, plus il sera possible d'adapter l'application dans des directions et selon des critères non anticipés au moment de son développement, entraînant une durée de vie accrue.

Cette analyse de la problématique des logiciels adaptatifs nous conduit à poser la thèse fondamentale qui est au cœur de nos travaux et de ce document : *nous considérons l'adaptation d'une application à son contexte d'exécution et aux évolutions de celui-ci comme un* aspect*, qui doit être développé séparément du reste de l'application en utilisant un formalisme spécifique, afin de pouvoir être intégré (ou retiré) dynamiquement dans le reste de l'application.*

Cette approche repose sur l'un des principes de base du génie logiciel, la Séparation de Préoccupations [Hürsch and Lopes, 1995] (*Separation of Concerns* en anglais). Ce principe général a pris de nombreuses formes au cours de l'histoire, parmi lesquelles la programmation modulaire [Parnas, 1972], la notion d'encapsulation qui a donnée naissance à la programmation par objet et à la programmation par composants, et plus récemment la *programmation par aspects* (AOP[6]) [Kiczales et al., 1997]. Cette dernière approche tente de trouver des solutions à un problème que les plus anciennes ne savent pas bien résoudre, à savoir l'entrelacement des préoccupations (*cross-cutting concerns*). Les autres approches, y compris la programmation par objet ou par composants, permettent de décomposer un système complexe en différents éléments plus simples à appréhender, mais cette décomposition est unique, et correspond à l'*une* des préoccupations présentes dans le système. Cette « tyrannie de la décomposition dominante » [Ossher and Tarr, 1999] empêche les autres préoccupations présentes dans le système d'être modularisées correctement dès lors qu'elles sont *transverses* par rapport à la décomposition dominante choisie. Celles-ci se retrouvent donc dispersées dans la structure du système (*cross-cut*) et sujettes à tous les problèmes du code non modulaire. Afin de résoudre ce problème, la programmation par aspects est censée permettre de modulariser correctement toutes les préoccupations présentes dans un système, même si elles sont transverses les unes par rapport aux autres. Pour cela, AOP introduit la notion de *tissage* (*weaving*), qui permet de composer correctement différents aspects (préoccupations transverses modularisées) en un système complet et cohérent.

[6]Aspect-Oriented Programming

La thèse énoncée plus haut considère l'adaptation comme un aspect. Cela signifie donc qu'il s'agit *(i)* d'une préoccupation séparée du reste du code « normal » de l'application (code métier), et *(ii)* que cette préoccupation est transverse par rapport au code métier, c'est-à-dire que sa modularisation ne suit pas la structure du code métier :

> « *While internal adaptation can certainly be made to work, it has a number of serious problems. First, when adaptation is interwined with application code it is difficult and costly to make changes in adaptation policy and mechanism. Second, for similar reasons, it is hard to reuse adaptation mechanisms from one application to another. Third, it is difficult to reason about the correctness of a given adaptation mechanism, because one must also consider all of the application-specific functionality at the same time. In contrast to the internal approach, externalized adaptation has many benefits : adaptation mechanisms can be more easily extended ; they can be studied and reasoned about independently of the monitored applications ; they can exploit shared monitoring and adaptation infrastructure.* » Garlan et al. [2001]

Étant donnée cette (hypo-)thèse, notre travail va donc consister à appliquer les idées de la conception et de la programmation par aspects à notre domaine spécifique, l'adaptation, et à prouver que cette approche est avantageuse par rapport à l'état de l'art.

Concrètement, notre objectif est de proposer aux programmeurs d'applications un modèle de développement, ainsi que les outils associés, leur permettant de développer plus facilement des applications capables d'être adaptées – voire de s'adapter elles-mêmes – aux différents environnements dans lesquels elles seront utilisées, et aux évolutions dynamiques de ceux-ci. Étant donnée l'approche que nous avons choisie, consistant à considérer l'adaptation comme un aspect et à appliquer les techniques développées dans les domaines de la Séparation des Préoccupations et de la Programmation par Aspects, cela implique de fournir aux programmeurs :

- les moyens techniques et méthodologiques pour mettre en œuvre cette séparation entre d'une part le code métier d'une application et d'autre part le code chargé d'adapter celui-ci ;
- des outils, sous forme de cadre de développement ou de langage dédié, pour le développement du code d'adaptation, prenant en compte sa nature spécifique ;
- les techniques et outils pour assembler (tisser) et/ou désassembler le code métier et le code d'adaptation aux moments appropriés, afin obtenir finalement des applications les plus adaptables et les plus autonomes possible.

### Limites

Tel que nous l'avons présenté, le problème de l'adaptation est très vaste. Il est bien évident que nous n'envisageons pas de résoudre *tous* les problèmes liés au développement d'applications adaptatives. En particulier, nous nous fixons certaines limites dans deux domaines : le type d'applications que nous voulons rendre adaptatives, et la nature des adaptations que nous prenons en compte.

Concernant le type d'applications, nous ne considérons que des applications nouvelles (non patrimoniales) développées à base de composants. Comme nous le verrons plus loin, notre proposition est basée sur un **modèle de composants** existant, Fractal [Bruneton et al., 2004], et plus particulièrement son implémentation de référence en Java. Cependant, les concepts et la méthodologie générale que nous proposons sont facilement transposables à d'autres modèles de composants et à d'autres langages similaires.

Pour ce qui est de la nature des adaptations supportées, c'est à dire des mécanismes utilisables pour transformer l'application, il existent de très nombreuses possibilités. D'ailleurs, toute technologie permettant de *modifier* une application peut, selon les circonstances de son utilisation, être considérée comme un mécanisme d'adaptation. Cependant, étant données d'une part la description faite plus haut de la motivation de nos travaux, qui met l'accent sur le besoin d'**adaptation dynamique**, et d'autre part le choix d'une approche à base de composants, nous ne considérerons comme mécanismes d'adaptation que les possibilités de **reconfigurations structurelles** de l'application en cours d'exécution.

Ces deux choix seront discutés et justifiés plus amplement aux chapitres 4 et 6.

## 1.2 Caractéristiques des logiciels adaptatifs

Dans cette section, nous définissons plus précisément la notion d'adaptation, en général et dans le cas du logiciel, puis nous étudions les caractéristiques spécifiques des logiciels adaptatifs et les critères qui nous servirons de grille de lecture dans le chapitre 3 qui étudie l'état de l'art.

### 1.2.1 Adaptation et logiciels adaptatifs

D'après le dictionnaire, *adapter* consiste à « rendre (un dispositif, des mesures, etc.) apte à assurer ses fonctions dans des conditions particulières ou nouvelles. » Cette définition implique la présence d'un *système* (le dispositif), qui réalise une *fonction* particulière. La façon dont le système réalise cette fonction dépend – au moins en partie – du *contexte* dans lequel il se trouve. Lorsque le contexte est modifié, créant des conditions particulières ou nouvelles, le système doit donc être *adapté* afin de continuer à réaliser sa fonction dans le nouveau contexte.

Une adaptation est donc une *modification* d'un système, en réponse à *un changement dans son contexte*, avec l'objectif que le système résultant soit mieux à même de réaliser sa fonction dans le nouveau contexte. Il est à noter que si toute adaptation est mise en œuvre par une modification du système, une modification quelconque d'un système *n'est pas* systématiquement une adaptation. Elle ne peut l'être que si le nouveau système, modifié, est *meilleur* que l'ancien, compte tenu des nouvelles conditions. Cette distinction est fondamentale, car elle met en évidence que l'on ne peut parler de l'adaptation d'un système que si l'on dispose de critères permettant d'évaluer sa *qualité*, qui détermine ce que « meilleur » signifie.

Plus formellement, étant donné un système $S$, on se donne une fonction $f$ (pour *fitness*, ou adéquation), qui évalue la *qualité* de $S$. En pratique, un système n'est jamais complètement isolé, et existe dans un contexte qui influence son fonctionnement. Si $C$ représente ce contexte, alors $f(S, C)$ représente l'adéquation de $S$ au contexte $C$. Si $S$ se trouve dans un contexte différent $C'$, il peut arriver que $f(S, C') < f(S, C)$, c'est-à-dire que $S$ est moins bien *adapté* à $C'$ qu'à $C$. Dans ce cas, nous devons donc modifier $S$ afin d'obtenir un nouveau système $S'$ tel que $f(S', C') \geq f(S, C')$[7]. Dans un système dynamique, le contexte évolue de façon continue au cours du temps. Ce processus d'adaptation doit donc lui aussi être renouvelé pendant toute la durée de vie du système, soit de façon continue, soit par étapes discrètes, suivant la nature du système.

Dans la suite de cette thèse, on dira qu'un système est *adaptable* s'il peut être adapté par une entité extérieure (logicielle ou non), et qu'un système est *adaptatif* s'il s'adapte automatiquement et de façon autonome. Un système adaptatif est à la fois le sujet et l'acteur de l'adaptation Ledoux et al.; Dowling and Cahill ; il se modifie donc lui-même, en fonction des évolutions de son contexte afin de toujours offrir la meilleure qualité possible relativement à ce contexte. Dans le cadre du génie logiciel, cette notion de système adaptatif correspond à un logiciel qui s'adapte dynamiquement – c'est-à-dire au cours de son exécution – aux caractéristiques et aux évolutions de son contexte d'exécution.

Un système adaptatif peut être caractérisé par :

- Un ensemble d'*opérations* permettant de modifier le système. Étant donné un état initial, cet ensemble d'opérations définit un espace d'états regroupant tous les états possibles du système. Dans le cas du logiciel, ces opérations se traduisent par des mécanismes de reconfiguration dynamique. Les caractéristiques de différents mécanismes possibles sont décrits et discutés dans la section 1.2.2.
- Un *contexte*, qui regroupe tous les éléments externes au système qui influencent son fonctionnement. Ce contexte est le plus généralement dynamique, évoluant de façon plus ou moins prévisible au cours du temps. Bien entendu, le système peut lui-même avoir une influence sur certains éléments de ce contexte. Ce point est approfondi dans le cas du logiciel à la section 1.2.3.
- Une *fonction d'adéquation*, qui permet à tout moment de savoir dans quelle mesure le système réalise ses objectifs relativement au contexte dans lequel il se trouve. Par définition, un système

[7] Le problème de l'adaptation est très proche de celui de l'optimisation, qui cherche à maximiser la fonction $f$ en modifiant le système, c'est-à-dire à déterminer le système $S_{max}$ tel que $\forall S, f(S) \leq f(S_{max})$. La différence est que l'optimisation se fait « à contexte constant », alors que le contexte, ses évolutions et son impact sur le système sont au cœur du problème de l'adaptation. On peut donc voir le processus d'adaptation comme une optimisation en continue, perpétuellement remise en cause (au moins partiellement) par les évolutions du contexte dans lequel fonctionne le système.

est « une combinaison de composants qui agissent ensemble afin de réaliser une fonction impossible à réaliser par aucune des parties » (définition de l'IEEE), et cette fonction est donc d'une certaine manière inhérente au système. En pratique, cette fonction est très rarement définie explicitement, et fait souvent appel à des critères plus ou moins subjectifs pour évaluer la qualité d'un système, qui peuvent dépendre de l'observateur. Dans notre cas particulier, cette fonction correspond à la spécification du problème pour lequel le logiciel à été conçu et aux critères de qualité de service.
– Une *stratégie d'adaptation*, chargée de mettre en œuvre le processus d'adaptation dans le système, afin qu'il soit toujours le plus performant possible (tel qu'évalué par la fonction d'adéquation), relativement à son contexte d'utilisation, et ce malgré les variations de ce dernier. Dans un logiciel, elle va se traduire par un ensemble d'algorithmes et de données de l'application adaptative chargés de réaliser les adaptations. Cette stratégie utilise pour cela les autres éléments du système : sa connaissance du contexte d'exécution, et l'existence de mécanismes de reconfigurations. La nature et les caractéristiques de cette stratégie sont développés dans la section 1.2.4.

### 1.2.2 Mécanismes de reconfiguration dynamique

La capacité d'un système adaptatif en général à être adapté est déterminée en grande partie par le nombre et la nature des opérations applicables au système, elles-mêmes en partie déterminées par la structure du système. Plus un système supporte d'opérations permettant de le modifier, et plus ces opérations sont « puissantes », plus il est souple et donc *a priori* capable d'être adapté à un grand nombre de situations.

Nous nous intéressons ici aux différents mécanismes existants permettant de reconfigurer dynamiquement une application. Il existe de très nombreux mécanismes différents permettant de reconfigurer une application. Il n'est bien entendu pas possible de les citer tous ici. Nous verrons dans le chapitre suivant comment un certain nombre d'entre eux sont appliqués dans les solutions existantes pour la construction d'applications adaptatives. Dans cette section, nous tentons plutôt de dégager un certain nombre de critères qui nous permettront d'évaluer ces solutions du point de vue des mécanismes utilisés. En effet, bien que tous ces mécanismes puissent être utilisés dans le cadre d'une adaptation, la plupart n'ont rien de spécifique à cette utilisation et peuvent être utilisés pour d'autres besoins. Cependant, les contraintes spécifiques de l'adaptation les rendent plus ou moins appropriés.

**Garanties.** Nous l'avons déjà dit plus haut, il est important de trouver le bon équilibre entre la puissance des reconfigurations possibles et le maintient de la cohérence de l'application. Les opérations de reconfigurations utilisées pour l'adaptation doivent donc offrir des *garanties* quand à leurs effets sur le système. Cela implique d'une part que l'effet de chaque opération soit *prévisible* et d'autre part que leurs définitions se basent sur une description plus ou moins *formelle* de la structure et du comportement du système. Ces deux sous-critères sont essentiels pour pouvoir raisonner *a priori* sur les effets des différentes opérations sur le système, c'est-à-dire sans avoir à les réaliser. On peut imaginer de nombreux types de garanties différentes, comme par exemple concernant les effets sur les performances de telle ou telle opération, mais la plus fondamentale, est la garantie du *maintient de la cohérence* du système adapté. Cette garantie stipule que quelles que soient les opérations appliquées au système, celui-ci continuera de fonctionner *correctement*. Il faut distinguer ici un système qui fonctionne correctement, c'est-à-dire qui produit les bons résultats sans erreur, d'un système qui fonctionne *bien*, c'est-à-dire qui, en plus d'être correct, utilise efficacement les ressources à sa disposition. Dans le cadre de l'adaptation, l'objectif est de faire en sorte qu'un système fonctionne au mieux, et donc le minimum que l'on doive garantir est qu'un système correct reste correct quelles que soient les adaptations qu'on lui applique. C'est ce que signifie le critère de maintient de la cohérence.

**Modularité.** La *modularité* représente le fait de pouvoir appliquer des modifications à une partie du système sans affecter les autres. Ce critère est un critère général du génie logiciel, qui n'est pas spécifique à l'adaptation. Cependant, il est particulièrement important dans notre cas, car il permet d'adapter des systèmes complexes en raisonnant sur leur propriétés locales, et donc sans avoir à se préoccuper

d'interactions non prévues avec l'extérieur. Cela signifie que si une adaptation est rendue nécessaire par une modification du contexte, par exemple une baisse de la qualité de connexion réseau, il est possible de n'adapter *que* les parties du système qui interagissent directement avec le réseau, en ayant un impact minimal (idéalement nul) sur le reste. Cela signifie aussi que l'on doit facilement adapter *tous* les éléments concernés. En effet, si dans l'exemple précédent du réseau on ne modifie que le côté client sans toucher au serveur, ceux-ci risquent de ne plus pouvoir communiquer. La possibilité d'effectuer des modifications de façon modulaire ne dépend pas uniquement de la nature des mécanismes utilisés et de leur granularité, mais aussi de la structure de l'application. Si l'application est mal modularisée, ou modularisée d'une façon incompatible avec une adaptation donnée, on retrouve ici le problème bien connu de la *tyrannie de la décomposition dominante* [Ossher and Tarr, 1999] qui ne peut être résolu que par des techniques de programmation par aspects.

**Performance.** La *performance* des reconfigurations est cruciale. Les mécanismes utilisés pour l'adaptation dynamique doivent pouvoir d'être implémentés efficacement, aussi bien en terme de vitesse d'exécution que d'utilisation de ressources. En effet, ils ne doivent pas perturber le fonctionnement de l'application plus que nécessaire. Un mécanisme qui nécessiterait plusieurs secondes pour s'exécuter serait inutilisable dans la plupart des applications. Même si le système résultant de son utilisation était beaucoup plus performant que le système initial, son coût serait prohibitif. Au delà des interférences avec l'exécution normale de l'application, une reconfiguration qui prend trop de temps risque de ne plus être valide une fois terminée si le contexte a évolué entre temps.

**Ouverture.** Il est en général impossible aux programmeurs d'une application de prévoir toutes les circonstances dans lesquelles elle sera utilisée, et donc les adaptations qu'elle devra supporter. Bien que cela ne soit pas à proprement parler nécessaire pour pouvoir parler d'application adaptative, en pratique il est important pour une telle application de supporter des *modifications non anticipées* au moment du développement. Un système fermé, qui ne supporterait que des adaptations prévues serait forcément limité, et deviendrait rapidement beaucoup trop complexe à développer s'il devait s'adapter à des scénarios réalistes.

**Transparence.** Enfin, notre dernier critère, lié au précédent, concerne la *transparence* des mécanismes du point de vue du programmeur initial de l'application. En effet, si l'on veut pouvoir adapter une application de façon non anticipée, il faut éviter des mécanismes qui nécessitent que le programmeur de l'application « prépare » l'application pour ces modifications. Il est important que la création d'applications adaptatives ne se fasse pas au dépend de la complexité du développement initial, déjà suffisamment difficile. Au contraire, notre objectif est plutôt de simplifier le travail des programmeurs en modularisant le code spécifique à l'adaptation dans un aspect, ce qui devrait rendre le code métier plus simple.

### 1.2.3 Contexte explicite

Bien que le contexte soit pas définition extérieur au système, un système adaptatif doit être conscient de celui-ci afin de pouvoir réagir à ses évolutions. Un système ne peut pas être adaptatif s'il est « aveugle » à son contexte. L'étendue et la précision – aussi bien spatiale (niveau de détail et d'exactitude) que temporelle (temps de latence) – des connaissances que possède le système sur son contexte est déterminant pour la qualité des adaptations qu'il peut mettre en œuvre. Si le système ne possède que des informations partielles, inexactes, ou obsolètes, les adaptations qu'il pourra réaliser au mieux seront non optimales, et au pire auront un effet négatif sur les performances du système.

Puisque l'adaptation d'un logiciel se fait par rapport à son contexte d'exécution et aux évolutions de celui-ci, il est primordial pour une application adaptative d'avoir *conscience* de ce contexte [Dey and Abowd, 2000]. Ce n'est pas le cas de la plupart des logiciels ; au contraire, les environnements d'exécution modernes (machines virtuelles, middlewares) essayent plutôt de les isoler le plus possible des spécificités de leur environnement. Si cette approche a de nombreux avantages, comme une meilleure portabilité, elle empêche parfois les applications de tirer partie des particularités de leur contexte d'exécution.

Le premier problème concerne l'*identification* du contexte du logiciel. La définition que nous utilisons (voir la section 5.1, page 71) indique que le contexte d'un logiciel regroupe tout ce qui influence la qualité de son fonctionnement, mais qui ne fait pas partie explicitement du logiciel. Ainsi, les caractéristiques matérielles de l'ordinateur hôte d'une application font partie de son contexte, mais pas les objets métiers qu'elle manipule directement. Dans ce document, nous ne nous intéressons pas à ce problème d'identification, mais seulement à la façon dont le contexte, une fois identifié, intervient dans l'adaptation du logiciel.

Un logiciel adaptatif doit donc contenir le code nécessaire pour le rendre *sensible* à son contexte. Les rôles de cette partie du logiciel sont :

- découvrir des caractéristiques du contexte spécifique à une exécution du logiciel ;
- détecter les changements significatifs qui se produisent dans le contexte d'exécution, et déclenchent éventuellement des adaptations du logiciel ;
- offrir au reste du logiciel une interface lui permettant d'obtenir ces informations.

Le chapitre 5 est spécifiquement consacré à cette problématique et à la solution que nous y proposons. Nous nous contenterons ici d'énumérer rapidement les critères les plus importants permettant d'évaluer cette partie d'un logiciel adaptatif :

**La précision des informations** obtenues, aussi bien *spatiale* que *temporelle*. La précision spatiale, correspond au niveau de détails fournis, et doit être la plus élevée possible. La précision temporelle correspond au décalage dans le temps entre le moment où les informations sont reçues par le logiciel afin d'être exploitées, et l'instant (passé) auxquelles elles se sont produites.

**La richesse** des informations correspond à l'étendue du contexte qui est effectivement observée par le système, et doit être la plus élevée possible.

**La généralité et la modularité** du code implémentant l'observation du contexte dans le logiciel, doit elle aussi être maximisée, afin de pouvoir être réutilisée dans d'autres applications. En effet, certains éléments, comme les caractéristiques matérielles ou logicielles de la plate-forme d'exécution font partie du contexte de tous les logiciels.

**Les performances** du code d'observation sont primordiales, afin d'éviter les interférences de cette partie du système adaptatif avec le fonctionnement normal du logiciel.

### 1.2.4 Stratégie d'adaptation

La stratégie d'adaptation employée par le système est d'une certaine façon chargée d'« incarner » la fonction d'adéquation sur le plan opérationnel. La fonction définit un objectif à atteindre, et la stratégie d'adaptation est chargée de réaliser cet objectif au mieux. La stratégie d'adaptation est l'élément le plus important d'un système adaptatif ; même si les opérations disponibles sont extrêmement puissantes et si le contexte est connu dans ses moindres détails, tout cela ne sert à rien si la stratégie n'est pas capable de tirer partie de ces informations.

Dans notre cas particulier, la stratégie d'adaptation d'un logiciel adaptatif se présente sous la forme d'un ensemble d'algorithme et de données qui sont chargés, à partir des informations connues sur le contexte d'exécution, de décider quand et comment adapter le logiciel, en utilisant au mieux les mécanismes de reconfiguration disponibles. La stratégie est donc au cœur du logiciel adaptatif, puisqu'elle fait le lien entre les deux autres éléments analysés plus haut : mécanismes de reconfiguration et informations contextuelles.

Voici les caractéristiques à notre sens les plus importantes d'une stratégie d'adaptation, qui nous serviront de critères d'évaluation. Notons que certains de ces critères sont des critères « classiques » du génie logiciel ; cependant, il sont tout particulièrement importants dans le cas d'une stratégie d'adaptation.

**Analysabilité.** La mise en œuvre d'une stratégie d'adaptation va conduire à des modifications dans l'application adaptée. Ces modifications sont *a priori* conçues pour améliorer l'application, mais rien dans les mécanismes utilisés ne peut empêcher ces modifications de nuire à la qualité du logiciel (les seules garanties que ces mécanismes peuvent nous offrir est de conserver la consistance du système). Il est donc important que la stratégie d'un logiciel adaptatif puisse être analysée avant

d'être exécutée afin de vérifier certaines propriétés (dépendant de l'application) et de garantir que la stratégie n'aura pas d'effets pervers, comme de rendre l'application inutilisable ou de générer des erreurs irrécupérables. Parmi les critères les plus importants concernant la qualité d'une stratégie d'adaptation, on trouve :

La stabilité. Il peut arriver que le contexte d'exécution du logiciel évolue de façon abrupte et totalement imprévisible. Une stratégie d'adaptation naïve risquerait par des réactions trop précipitées ou disproportionnées de rendre le système instable.

L'agilité. Le temps de réaction de la stratégie par rapport aux évolutions du contexte doit être le plus faible possible, non seulement afin de ne pas perturber le fonctionnement normal du logiciel, mais aussi car si la stratégie met trop de temps à se décider, le contexte peut avoir suffisamment évolué entre temps pour rendre sa décision caduque.

**Dynamicité.** Tout comme pour les mécanismes de reconfigurations, il est important que la stratégie d'adaptation d'un logiciel soit ouverte, afin de pouvoir elle-même être adaptée et étendue de façon non anticipée sans avoir à stopper, modifier, puis recompiler le logiciel.

**Généricité et réutilisabilité.** De la même manière qu'elles partagent une partie de leur contexte (en particulier en ce qui concerne les caractéristiques matérielles des plates-formes hôtes), beaucoup d'applications utilisent des services non-fonctionnels de type middleware similaires, voire identiques. Les stratégies d'adaptation de ces couches logicielles sont relativement indépendantes de la fonction spécifique du logiciel et doivent donc pouvoir être réutilisées, au moins en partie, dans plusieurs applications adaptatives.

## 1.3 Conclusion

Dans ce chapitre, nous avons tout d'abord présenté une certain nombre de problèmes posés par différents types de variabilité (spatiale, temporelle, des besoins), qui rendent de plus en plus difficile la création de nouvelles applications dans le contexte technologique actuel. Nous arguons que la création d'*applications adaptatives* permet de résoudre – ou au moins d'atténuer – ce problème, mais qu'aucune technologie ni méthodologie n'existent actuellement pour traiter ce problème spécifique. Nous avons donc définit l'objectif de cette thèse comme étant de *faciliter le développement d'applications adaptatives*, dans certaines limites bien identifiées. Notre approche pour cela sera de considérer l'adaptation comme un *aspect*, au sens de la programmation par aspects, et d'utiliser, sans s'y limiter, les techniques de ce domaine : modularisation de l'aspect par rapport au code applicatif, et intégration (tissage) dynamique.

Dans la seconde partie de ce chapitre, nous avons effectué une première analyse de la notion d'adaptation appliquée au logiciel, et des différents éléments qui constituent un logiciel adaptatif : sensibilité au contexte, présence de mécanismes de reconfiguration dynamique, et réification des stratégies d'adaptation. Cette analyse nous a permis d'identifier pour chacun de ces éléments un certain nombre de critères d'évaluation qui nous serviront par la suite, aussi bien pour évaluer les propositions existantes que pour guider la conception de notre propre solution.

# Chapitre 2

# Concepts et technologies sous-jacents à notre proposition

**Sommaire**



DANS CE COURT CHAPITRE, nous présentons rapidement un certain nombre de domaines de recherche liés à nos travaux. La séparation des préoccupations, et plus spécifiquement la programmation par aspects (Section 2.1) sont au cœur de notre approche de l'adaptation, tout comme la notion de composant logiciel (Section 2.3). La réflexion (Section 2.2) est plus un outil qui nous permettra de mettre en œuvre les mécanismes dont nous avons besoin sans avoir recours à des techniques *ad hoc*. Cette introduction à la réflexion est aussi importante pour comprendre un certain nombre des travaux étudiés dans le chapitre suivant. La notion de langage dédié présentée dans ce chapitre (Section 2.4) est aussi utilisée par un certain nombre de ces travaux, mais sera surtout utilisée dans la suite de ce document pour exprimer l'aspect d'adaptation.

## 2.1 Séparation des préoccupations et programmation par aspects

> Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. [. . . ] It is what I sometimes have called « *the separation of concerns* », which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of.
>
> — On the role of scientific thought (E. W. Dijkstra)

### Présentation

La *séparation des préoccupations* (*Separation of Concerns*) est un des principes fondamentaux du génie logiciel, et un des meilleurs moyens dont nous disposons pour maîtriser la complexité des logiciels. Tout logiciel fait intervenir un certain nombre de *préoccupations* (concepts, sujets ou objectifs). Les interactions entre ces différentes préoccupations sont en grande partie responsables de la complexité des

logiciels. Si certaines de ces interactions sont intrinsèques au problème à résoudre, d'autres sont contingentes (accidentelles), et n'existent que parce que la *structure* du logiciel n'est pas en accord avec celle des préoccupations. Le principe de séparation des préoccupations dit tout simplement que *la structure du logiciel doit refléter le plus fidèlement possible les relations intrinsèques entre les préoccupations impliquées* afin de minimiser ces interactions accidentelles et de faciliter l'expression et la compréhension des interactions intrinsèques.

Parnas est le premier à avoir démontré l'importance de la séparation des préoccupations, à l'époque dans le cadre de la programmation modulaire. Dans [Parnas, 1972], il décrit deux conceptions possibles d'une même application. Toutes les deux utilisent les mêmes techniques de programmation modulaire, mais diffèrent par les critères choisis pour effectuer la modularisation. La première conception décompose l'application en fonction des différentes tâches successives qu'elle doit effectuer ; chaque module représente une *phase* distincte du traitement. La seconde isole chaque *décision de conception*, concernant soit la représentation des données soit les algorithmes utilisés, dans un module différent : « *Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.* » [Parnas, 1972].

Parnas montre que bien que les deux décompositions donnent des programmes corrects, la seconde est bien meilleure en terme d'évolutivité, permet le développement séparé des différents modules et est plus simple à appréhender et donc à maintenir. L'utilisation comme critère de modularisation des *choix de conception* auxquels est confronté le programmeur permet de faire correspondre chaque module du programme à une préoccupation. En effet, si le choix effectué par le programmeur concernant, par exemple, l'algorithme utilisé pour implémenter telle ou telle fonctionnalité peut être différent, le fait même qu'il se pose la question « Comment implémenter cette fonctionnalité ? » indique que la fonctionnalité en question est une *préoccupation* du programme. Pour Parnas, chaque choix auquel est confronté le programmeur, et donc chaque module, correspond ainsi à une préoccupation.

Bien que ce principe nécessite pour être mis en œuvre l'existence d'une ou plusieurs technologies permettant de structurer le logiciel, la séparation des préoccupations *n'est pas* une technologie. C'est un *principe de conception* qui nous guide afin d'utiliser au mieux les technologies disponibles. Ce principe s'est incarné successivement dans toutes les techniques de programmation, chacune permettant d'aller plus loin dans l'isolation des différentes préoccupations. Il est intéressant d'observer l'évolution des techniques de conception et de programmation du point de vue de la recherche de mécanismes de séparation des préoccupations de plus en plus efficaces :

- la programmation modulaire permet de regrouper un ensemble de fonctions qui traitent d'une même préoccupation ;
- la programmation par objets va plus loin en encapsulant la représentation des données manipulées par ces fonctions ;
- la programmation par composants pousse la séparation encore plus loin. Les modèles de composants renforcent la notion de *contrat* entre les différents éléments du logiciel. Un contrat peut être vu comme un type étendu (au delà de la simple liste de signatures), toutes les interactions entre composants reposant uniquement sur l'interface décrite dans le contrat. Les différents composants sont ainsi entièrement *isolés* et tant qu'ils respectent l'interface, ils peuvent être implémentés dans des langages différents, voire s'exécuter sur des machines différentes.

Chacune de ces techniques permet de séparer de plus en plus les différents éléments constituant un logiciel, leurs interactions étant limitées au strict minimum nécessaire. Idéalement, chacun de ces éléments (module, classe/objet, composant) devrait correspondre à une et une seule préoccupation. Malheureusement, cette isolation de plus en plus stricte, « en profondeur », n'est pas toujours suffisante. La programmation par aspects est apparue depuis quelques temps comme un complément à ces techniques permettant une séparation « en largeur ».

Les techniques et concepts évoqués ci-dessus ont ceci de commun qu'elles décomposent un logiciel de façon hiérarchique, selon une et une seule dimension. Malheureusement la réalité n'est pas toujours aussi simple, et les préoccupations qui interviennent dans la modélisation d'un logiciel correspondent rarement à une telle structure. La présence des seuls mécanismes de décomposition hiérarchique conduit

alors à ce que Tarr et Ossher appellent la « tyrannie de la décomposition dominante » [Tarr et al., 1999]. Les concepteurs choisissent une préoccupation principale – typiquement une préoccupation métier – selon laquelle ils décomposent le logiciel sous forme de modules[1]. L'unique dimension disponible ayant été monopolisée par la préoccupation dominante, la plupart des autres préoccupations qui sont ensuite intégrées au logiciel ne peuvent alors plus être modularisées proprement. On parle de *préoccupations transverses* (*cross-cutting concerns*) pour désigner ces préoccupations qui se retrouvent dispersées dans le reste du code (*code scattering*) et mélangées (*code tangling*) avec les autres. Ces caractéristiques rendent le logiciel difficile à comprendre, à faire évoluer et à maintenir.

Certaines de ces préoccupations transverses sont des préoccupations « métier » complémentaires de la préoccupation dominante choisie, mais très souvent il s'agit de préoccupations dites non-fonctionnelles, qui traitent par exemple de la sécurité (flots de données), des performances (politique de gestion mémoire), ou bien encore de la synchronisation. Cela est d'autant plus vrai que ces préoccupations n'étant pas liées à une application donnée, elles sont souvent implémentées sous forme de bibliothèques ou de *frameworks* réutilisables, dont la décomposition, imposée, est forcément incompatible avec celle de l'application[2].

La notion d'*aspect* a donc été proposée pour désigner une préoccupation modularisée bien que transverse par rapport une décomposition donnée. Bien entendu, cette modularisation auparavant impossible nécessite la création de nouveaux mécanismes de programmation :

> « When writing a modular program to solve a problem, one first divides the problem into sub-problems, then solves the sub-problems and combines the solutions. The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase ones ability to modularise a problem conceptually, one must provide new kinds of glue in the programming language. » [Hughes, 1989]

La *programmation par aspects* (AOP[4]) [Kiczales et al., 1997; Elrad et al., 2001; Bouraqadi-Saâdani and Ledoux, 2001] est un domaine de recherche dont l'objectif est d'incorporer aux langages de programmation existants (qu'ils soient procéduraux, fonctionnels, objets ou autre) de nouvelles constructions pour permettre la définition d'aspects de façon modulaire.

Bien que d'autres pistes et d'autres formes soient étudiées, la plupart des propositions actuelles pour la programmation par aspects sont caractérisées par :

- Une dissymétrie entre d'une part le programme dit « de base », qui correspond à la décomposition principale, et d'autre part les aspects, spécifiés séparément et de façon modulaire[5].
- L'ajout au langage de deux nouvelles constructions :
  - un langage de coupe (*pointcut language*), qui permet de désigner de façon déclarative un ensemble de points de jonctions (*join points*) se trouvant dans le programme de base. Ces points de jonction peuvent correspondre soit à des éléments du code source du logiciel (point d'entrée d'une méthode par exemple [Kiczales et al., 2001]), soit à des événements se produisant lors de l'exécution (appel de méthode [Douence et al., 2002]).
  - et un langage d'actions (*advices*), qui permet d'associer à chaque coupe l'action (éventuellement complexe) correspondante. La plupart des propositions actuelles utilisent pour le langage d'action le langage de programmation de base (par exemple Java)[6].

Un aspect est alors un module qui regroupe et intègre des couples $(coupe, action)$ qui ensemble implémentent une préoccupation, en modifiant la sémantique du programme de base de façon transparente (pour le programmeur de base) [Filman and Friedman, 2000]. Le programme de base et les aspects sont ensuite *tissés* (*weaved*) en un tout cohérent, soit statiquement (par transformation de code source [Kic-

[1] « Module » est ici à prendre au sens général, cela peut être des objets ou des composants.

[2] Les conteneurs d'applications comme les serveurs EJB[3] [DeMichiel, 2003] par exemple permettent d'intégrer de façon déclarative certaines préoccupations non-fonctionnelles dans le code métier. Cependant, ces solutions sont conçues spécifiquement pour traiter un ensemble fixe de préoccupations (persistance, transactions...) [Jarir, 2002]. L'émergence de la programmation par aspects est en train de faire évoluer ces systèmes vers des solutions plus générales [Burke].

[4] Aspect-Oriented Programming

[5] Voir cependant le système HyperSpaces [Ossher and Tarr, 1999] qui ne fait pas cette distinction et dans lequel toutes les préoccupations sont symétriques.

[6] Les premières versions d'AspectJ utilisaient des langages spécifiques pour chaque aspect (*Aspect Specific Languages*).

zales et al., 2001]), soit dynamiquement (typiquement par le biais de la réflexion) [Douence et al., 2002; Pawlak et al., 2001].

Bien que la programmation par aspects apporte de nouvelles possibilités de structuration des logiciels, elle pose aussi un certain nombre de nouveaux problèmes. En effet, la non-localité des aspects par rapport au programme de base rend la compréhension de ce dernier difficile, puisque sa sémantique réelle n'est pas visible complètement ; ce problème peut cependant être mitigé par l'utilisation d'environnements de programmation adéquats[7]. Un autre problème, qui lui aussi découle de la non-localité des aspects, est la possibilité de conflits entre plusieurs aspects, par exemple lorsqu'ils modifient les mêmes points de jonction dans le programme de base. Seules des approches formelles, comme par exemple EAOP[8] [Douence et al., 2002] permettent de détecter et de résoudre ce genre de conflits.

#### Lien avec l'adaptation

On a déjà vu que rendre une application adaptative implique d'y intégrer du code pour : *(i)* observer l'environnement et détecter les changements significatifs qui s'y produisent, *(ii)* décider des reconfigurations appropriées, et enfin *(iii)* appliquer ces décisions. Chacun des ces éléments pris séparément est une préoccupation, tout comme l'est leur coordination en un tout cohérent, qui représente la *stratégie d'adaptation* de l'application. Or, aucune de ces préoccupations ne s'intègre dans la structure d'une application :

1. L'observation de l'environnement est par nature asynchrone par rapport à l'exécution de l'application et le code correspondant ne peut donc pas être intégrée dans la structure du logiciel (sauf dans le cas particulier des serveurs événementiels construits autour d'une boucle de réaction à des événements externes).
2. La prise de décision, qui est déclenchée par la détection (asynchrone) d'événements significatifs, peut nécessiter d'observer l'état de nombreux éléments dispersés dans l'application.
3. Enfin, les modifications à mettre en œuvre peuvent concerner là aussi des éléments divers qui ne sont pas forcément regroupés dans la structure du logiciel.

On voit donc que la structure du code correspondant à la stratégie d'adaptation d'un logiciel ne correspond pas en général à celle du code métier. L'adaptation est donc une préoccupation transverse, que l'on ne peut modulariser qu'en utilisant des techniques de programmation par aspects. Outre le fait de rendre le code correspondant plus simple à développer et à comprendre, la modularisation de l'adaptation a l'avantage de découpler le code métier de la stratégie d'adaptation. Si ces deux aspects étaient tissés statiquement, le code métier deviendrait inutilisable dans des contextes qui n'ont pas été prévus explicitement dans la stratégie. En permettant de spécifier la stratégie dynamiquement, on rend le code métier beaucoup plus réutilisable.

L'idée fondamentale de nos travaux est donc d'appliquer les principes de la séparation des préoccupations et de la programmation par aspects pour rendre modulaire et dynamique la stratégie d'adaptation des logiciels, avec tous les bénéfices que cela implique.

## 2.2 Réflexion et méta-programmation

#### Présentation

Un logiciel réflexif [Smith, 1984] est un logiciel qui possède une auto-représentation décrivant la connaissance qu'il a de lui-même, et capable en modifiant cette représentation de se modifier. Un tel logiciel est donc capable de raisonner et d'agir sur lui-même.

Dans un logiciel réflexif, on distingue la capacité d'*introspection*, qui correspond à la possibilité de s'observer, de la capacité d'*intercession*, qui permet de s'auto-modifier. L'introspection permet au logiciel de raisonner sur lui-même et de répondre à des questions le concernant, et est basée sur une *réification* des

[7] Voir par exemple *AspectJ Development Tools* : `http://www.eclipse.org/ajdt`
[8] Event-based Aspect-Oriented Programming

structures et des mécanismes du logiciel qui sont habituellement implicites. L'intercession, c'est-à-dire la modification du logiciel par lui-même, est possible lorsque cette réification peut elle-même être modifiée, et que ces modifications se reflètent immédiatement sur le logiciel. On parle alors de *connexion causale* [Maes, 1987] pour indiquer que la réification n'est pas une simple « photographie » de l'état du système à un instant donné, mais détermine effectivement le comportement du logiciel.

On distingue deux aspects complémentaires de la réflexion :

- La *réflexion structurelle* permet d'observer et de manipuler la façon dont le logiciel est organisé. Dans le cas d'un logiciel écrit dans langage à objet cela correspond à la découverte dynamique de la classe d'un objet, des variables d'instances et des méthodes dont il dispose, et à la modification de ceux-ci (par exemple pour changer la valeur d'une variable d'instance).
- La *réflexion comportementale* s'intéresse à l'exécution du programme. Elle permet par exemple de savoir quelles sont les méthodes ou fonctions en cours d'exécution, d'invoquer une méthode sur un objet donné ou de modifier la sémantique de certaines constructions du langage.

Dans un logiciel réflexif, on parle de niveau *de base* pour désigner le code « normal », c'est-à-dire non réflexif, qui traite du sujet principal du logiciel (des comptes bancaires, des images, du son...) et de *niveau méta* (ou parfois de méta-niveau) pour désigner le code réflexif, qui manipule les réifications du logiciel lui-même[9] (objets, classes, méthodes, messages...).

Grâce à l'abstraction des données (encapsulation) et à leur organisation (héritage, composition), les langages à objets sont particulièrement adaptés à la mise en œuvre d'architectures réflexives [Maes, 1987]. Les objets du niveau méta, appelés *méta-objets*, décrivent la représentation et contrôlent le comportement des objets du niveau de base. Les protocoles à méta-objets (MOP[10]) règlent la communication entre les objets et les méta-objets [Kiczales et al., 1991], et constituent donc l'interface entre les différents niveaux. Il est alors possible de modifier les méta-objets standards pour introduire de nouvelles sémantiques de représentation et d'exécution des objets (concurrence, localisation des objets répartis, envoi de messages distants [McAffer, 1995]). On peut voir les protocoles à méta-objets comme l'application du principe de l'*Open Implementation* [Kiczales, 1996] au niveau des langages à objets eux-mêmes. Le MOP de CLOS[11] est sans doute l'exemple le plus représentatif de cette approche [Kiczales et al., 1991].

En ouvrant l'implémentation du langage lui-même aux programmeurs d'applications, la réflexion leur fournit un outil très puissant pour faire évoluer ce langage afin de le rendre mieux *adapté* aux besoins spécifiques de leur application. En effet, la conception d'un langage de programmation généraliste nécessite de nombreux compromis au niveau des mécanismes supportés ou non et de leur implémentation afin de satisfaire le plus grand nombre. Lorsque l'on développe une application particulière, certains de ces choix peuvent se révéler inadaptés. La réflexion permet de remettre en cause certains de ces choix et de créer relativement simplement un nouveau langage mieux adapté aux besoins spécifiques d'un domaine ou d'une application [Kiczales et al., 1991]. Malheureusement, cette puissance a souvent un coût prohibitif, d'une part en terme de sécurité (il devient possible, et même facile de créer des systèmes incohérents) et d'autre part en terme de performances (puisque tout dans le système doit pouvoir être modifié à tout moment, la plupart des optimisations habituelles sont impossibles[12]).

### Lien avec l'adaptation

Malgré ces problèmes potentiels, la réflexion est un outil très important pour la réalisation de logiciels adaptatifs – ou même simplement adaptables. En effet, l'une des raisons pour lesquelles on a besoin d'adaptation est qu'il n'est pas toujours possible de prévoir lors du développement les circonstances dans lesquelles un logiciel sera utilisé. Dans ce cas, les stratégies d'adaptations ne peuvent pas être encodées statiquement dans le logiciel, puisqu'elles ne seront connues que lors pendant l'exécution. Non seulement la réflexion permet de modifier dynamiquement la structure et le comportement des applications,

[9] Le niveau méta étant lui-même un programme, il est possible d'introduire un *méta méta niveau*, et ainsi de suite à l'infini (conceptuellement). Une telle architecture, constituée de plusieurs couches (étages) de plus en plus abstraites, est nommée *tour réflexive*.

[10] Metaobject Protocol

[11] Common Lisp Object System

[12] Voir cependant [Tanter et al., 2003] qui permet une sélection spatiale et temporelle fine des mécanismes de réification.

mais puisqu'un méta-programme manipule directement les mécanismes du langage de programmation, il est indépendant des détails spécifiques à une application donnée. La réflexion permet donc de créer des stratégies d'adaptations *dynamiques*, et *génériques*, qui pourront être utilisées potentiellement sur n'importe quel programme. Bien entendu, la puissance de la réflexion permet aussi de rendre une application inutilisable. Tout l'enjeu consiste à trouver le bon niveau d'ouverture pour permettre des adaptations non-anticipées tout en garantissant le bon fonctionnement du système.

## 2.3 Composants logiciels

### Présentation

La notion de *composant logiciel* est apparue pour la première fois dans [McIlroy, 1968] où l'auteur décrit sa vision d'un « marché » de composants logiciels *réutilisables* et facilement combinables afin de construire rapidement des systèmes complexes. Sur le plan purement technique (nous ne nous intéressons pas ici aux aspects socio-économiques) de nombreuses définitions de la notion de composant logiciel ont été proposées, mais celle qui semble la plus couramment acceptée est donnée par [Szyperski, 1997] :

> « Un composant est une unité de composition dont les interfaces et les dépendances contextuelles sont spécifiées sous forme de contrats explicites. » [13]

L'objectif principal de la programmation par composants est de faciliter la réutilisation de code sous la forme de composants logiciels génériques. Grâce à une description explicite des services offerts (interfaces) et requis (dépendances contextuelles), de tels composants « sur l'étagère » (COTS[14]) peuvent être plus facilement distribués, puis intégrés – éventuellement après configuration – dans d'autres applications. On peut voir un composant comme un fragment de code (bibliothèque, classe, *framework*...) associé à des méta-données qui explicitent (sous forme de contrats) la manière dont il s'utilise, se configure, s'étend...

De nombreuses technologies ont été proposées pour réaliser cet objectif de composants sur l'étagère. On peut citer pour les modèles industriels : COM (*Component Object Model*) de Microsoft (avec ses variantes), CCM[15] [Group, 2002] du consortium OMG[16], les EJBs de Sun [DeMichiel, 2003] et plus récemment Fractal [Bruneton et al., 2003]. Sur le plan académique, citons l'approche ADLs[17] [Medvidovic and Taylor, 2000] qui se concentre sur la description de la composition (architecture) plutôt que sur la construction des composants individuels, Jiazzi [McDirmid et al., 2001] et ArchJava [Aldrich et al., 2002] qui étendent le langage Java et le modèle RM-ODP[18] [Blair and Stefani, 1998].

### Lien avec l'adaptation

Dans tous les cas, la programmation par composants offre des mécanismes permettant de réduire le couplage entre les différents éléments constituant une application (séparation forte entre l'interface d'un composant et son implémentation), et supporte une notion explicite d'*architecture* permettant de structurer les applications. Ces caractéristiques en font l'approche privilégiée pour la réalisation d'*applications adaptatives* : d'une part, le couplage lâche garantit que les modifications locales nécessaires à la réalisation d'une adaptation n'auront pas ou peu d'impact sur le reste de l'application, et d'autre part l'architecture décrite explicitement par le programmeur et/ou l'assembleur offre les points d'ancrage nécessaires à la réalisation d'opérations de reconfiguration.

Dans l'approche que nous avons choisie, l'adaptation d'une application se traduit par une reconfiguration dynamique – à l'exécution – de la structure/architecture de cette application. Le comportement de l'application étant déterminé par sa structure (le code source, bien que statique, détermine le comportement du logiciel), ces modifications structurelles permettent d'adapter le comportement de l'application [Oreizy et al., 1998, 1999].

[13] « *A component is a unit of composition with contractually specified interfaces and explicit context dependencies only.* »
[14] Components Off The Shelf
[15] CORBA Component Model
[16] Object Management Group
[17] Architecture Description Languages
[18] Reference Model for Open Distributed Processing

Un des avantages des modèles de composants est de fournir des abstractions plus riches que la plupart des langages de programmation pour représenter les relations existant entre les différents éléments (composants) d'un logiciel, et surtout de préserver ces relations jusqu'à l'exécution [Aldrich et al., 2002] : dans une application construite avec des composants, les différents types de relations possibles entre composants – connexion, dépendance, contenance, etc. – sont manifestes. Pour pouvoir spécifier et exécuter les reconfigurations structurelles nécessaires à l'adaptation d'une application, il faut que la structure de cette dernière soit à la fois *explicite* et *manipulable*. Autrement dit, ce dont nous avons besoin pour pouvoir construire des applications adaptatives, c'est un *modèle de composants réflexif*, qui permette à la fois l'introspection (architecture explicite [Cazzola et al., 1998]) et l'intercession (manipulation) de la structure de l'application en cours d'exécution.

## 2.4 Langages dédiés

**Présentation**

La plupart des langages de programmation utilisés par les développeurs (C, Java, Lisp, Haskell...) sont des langages *généralistes*, c'est-à-dire qu'ils sont conçus pour pouvoir créer n'importe quel type de logiciel, quel que soit le domaine. À l'inverse, les langages dédiés, ou DSLs[19] sont des langages conçus spécifiquement pour un domaine d'application particulier. La plupart du temps (mais pas toujours), ces langages dédiés ne sont pas aussi puissants que les langages généralistes (au sens où ils ne sont pas Turing-complets).

[van Deursen et al., 2000] propose la définition suivante de la notion de langage dédié :

> « Un langage dédié est un langage de programmation ou un langage de spécification exécutable qui offre, grâce à des notations et abstractions appropriées, un pouvoir d'expression concentré sur, et généralement limité à, un domaine d'applications particulier. » [20]

Les avantages des langages dédiés par rapport aux langages généralistes sont les suivants :

- Le pouvoir d'expression généralement limité du langage permet d'offrir des garanties et des analyses statiques plus poussées que dans le cas des langages généralistes. Par exemple, le langage d'interrogation de bases de données SQL[21] n'est pas récursif, ce qui garantit que les requêtes SQL terminent toujours.
- L'utilisation d'un niveau d'abstraction et de notations spécifiques au problème rend le code beaucoup plus lisible et concis. La définition d'une grammaire basée sur la syntaxe BNF[22] est beaucoup plus simple à écrire, à comprendre et surtout à faire évoluer qu'un analyseur syntaxique écrit « manuellement », même en utilisant des idiomes de programmations appropriés.
- Le développement de programmes à l'aide de DSLs est généralement beaucoup plus rapide qu'avec un langage généraliste, et le résultat a plus de chances d'être correct. En effet, ces langages sont souvent plus déclaratifs qu'impératifs, et leur niveau d'abstraction élimine un grand nombre d'erreurs courantes dans des langages de plus bas niveau (gestion de la mémoire, « *buffer overflows* », etc.).
- Les concepts manipulés par un langage dédié correspondent aux connaissances des experts dans ce domaine et les rendent réutilisables plus facilement par d'autres, qu'ils soient programmeurs ou non. Ainsi, des langages/outils comme Lex et Yacc « encapsulent » l'état de l'art en ce qui concerne la définition et la reconnaissance de langages par des grammaires régulières.
- Les programmes écrits dans un langage dédié peuvent souvent avoir de meilleures performances que ceux écrits dans des langages généralistes, car les implémentations de ces langages peuvent utiliser les meilleurs algorithmes connus et tirer partie d'optimisations spécifiques à leur domaine. Ainsi, les

[19] Domain-Specific Languages

[20] « *A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.* »

[21] Structured Query Language

[22] Backus-Naur Form

bases de données commerciales implémentent de nombreuses optimisations parfois très complexes (analyses de plan, réordonnancement...) sur les requêtes SQL fournies par leurs utilisateurs.

Cependant, les DSLs ont aussi un certain nombre d'inconvénients :

- Ils sont difficiles à concevoir et à implémenter, car ils requièrent un double-expertise assez rare, aussi bien dans le domaine concerné que dans celui de l'implémentation de langages de programmation.
- Au delà des problèmes d'implémentation, la conception même d'un DSL peut être très difficile, car il n'est pas simple de trouver le bon niveau d'abstraction et de généralité pour le langage. Un langage trop généraliste risque de ne pas tirer partie de tous avantages potentiels des DSLs. D'un autre côté, un langage trop restrictif risque de ne pas être utilisé s'il n'inclut pas toutes les constructions nécessaires.
- À chaque fois qu'un nouveau langage dédié est créé, ses futurs utilisateurs doivent être formés à son utilisation. L'apprentissage d'un nouveau langage, surtout spécialisé, est plus long et difficile que celui d'une bibliothèque équivalente dans un langage généraliste déjà connu. De même, si une entreprise utilise un langage généraliste comme C ou Java, elle a à sa disposition des centaines de milliers de programmeurs formés à l'utilisation de ce langage. Si elle utilise un langage dédié, il se peut que seuls quelques centaines, voire dizaines de personnes le connaisse.

Ces problèmes rendent difficile la décision de développer un nouveau DSL dans un cas particulier. Pour cette raison, la plupart des DSLs commencent leur vie sous la forme de bibliothèques ou de *frameworks*, sur lesquels sera éventuellement greffée une syntaxe si le besoin s'en fait sentir. La bibliothèque devient alors l'exécutif (*runtime*) du nouveau langage. Il arrive aussi qu'au lieu de créer un nouveau langage complet, un langage existant soit étendu de façon minimale pour intégrer de nouvelles constructions. De ce point de vue, les nouvelles constructions qu'AspectJ [Kiczales et al., 2001] ajoute à Java sont des DSLs, dont le domaine est ici la définition de coupes (*pointcuts*) et d'aspects. Voir aussi [Briot and Guerraoui, 1996] pour un exemple des différents niveaux d'intégration possibles (applicatif, intégré, réflexif), ici dans le cas particulier de la conception de systèmes à objets parallèles et répartis.

Le développement d'un langage dédié n'est pas très différent techniquement de celui d'un langage généraliste, pour lequel il existe une littérature très complète. Certains DSLs sont implémentés sous la forme d'interprètes, de compilateurs, ou par des techniques de génération de code (macros Lisp, *Generative Programming* [Czarnecki, 1998]). La principale différence repose dans l'étude de domaine préalable [Consel and Marlet, 1998] qui doit être conduite afin de déterminer le bon niveau d'abstraction et de généralité, les concepts et les notations appropriées.

### Lien avec l'adaptation

Le domaine spécifique qui nous occupe dans ce document est l'adaptation dynamique des applications aux évolutions de leur contexte d'exécution. Nous avons déjà justifié dans le chapitre d'introduction (1, page 3) et dans la section 2.1 que ce domaine peut être considéré comme un aspect, et à ce titre développé de façon modulaire et séparée du code applicatif. L'objectif final de nos travaux est de rendre plus aisé le développement d'applications adaptatives, et donc en particulier de ce code d'adaptation. Étant donnés les avantages des DSLs décrits ci-dessus, nous pensons que le développement de langage(s) dédié(s) à la spécification de stratégies d'adaptation est exactement ce dont nous avons besoin pour atteindre cet objectif. L'analyse des caractéristiques des logiciels adaptatifs effectuée dans la section 1.2 est une première ébauche de l'étude de domaine nécessaire à la création d'un tel langage (ou d'un ensemble de langages), qui sera développée et approfondie tout au long de ce document. Quant aux autres désavantages des langages dédiés (conception et implémentation complexe, apprentissage nécessaire), ils sont atténués par le fait que l'adaptation, bien qu'étant un domaine spécifique, est aussi une préoccupation commune à un très grand nombre de logiciels, comme nous l'avons montré dans la section 1.1.

# Chapitre 3

# État de l'art

## Sommaire



Ce chapitre présente et évalue un certain nombre de travaux qui représentent actuellement l'état de l'art dans le domaine du logiciel adaptatif.

## 3.1 Introduction et rappel de nos critères d'évaluation

L'objectif de ce chapitre est de présenter et d'évaluer des propositions existantes pour le développement de logiciels adaptatifs. Chacune des propositions sélectionnées est tout d'abord décrite, puis évaluée en fonction des critères que nous avons identifié comme étant pertinents pour les logiciels adaptatifs (voir la section 1.2, page 10). Avant d'étudier les différentes propositions, rappelons succinctement ces critères.

Concernant les mécanismes de reconfiguration :

**Garanties :** L'effet de chaque opération de reconfiguration est-il déterministe et bien défini ? Les opérations de reconfiguration permettent-elles de garantir que l'intégrité du logiciel est conservée ?

**Modularité :** Est-il possible d'appliquer une reconfiguration à une partie du système sans affecter le reste du système ? Est-il possible d'identifier les parties d'un système qui dépendent de tel ou tel élément du contexte, afin de pouvoir effectuer des reconfigurations globalement consistantes ?

**Performance :** Les mécanismes de reconfiguration supportés peuvent-ils être implémentés de façon efficace afin de ne pas perturber le fonctionnement normal du logiciel ?

**Ouverture :** Est-il possible de reconfigurer le logiciel d'une façon qui n'a pas été prévue explicitement lors de sa conception ?

**Transparence :** Le support de ces mécanismes par le logiciel requiert-il un effort important de la part des programmeurs du logiciel ?

Concernant la conscience qu'a le système logiciel de son contexte d'exécution :

**Précision :** Les informations disponibles concernant le contexte d'exécution sont-elles suffisamment détaillées et à jour pour permettre de prendre les bonnes décisions d'adaptation ?

**Richesse :** Ces mêmes informations couvrent-elles tous les aspects significatifs du contexte d'exécution ?

**Généralité :** Le code responsable de l'observation du contexte est-il suffisamment générique pour être réutilisé dans différentes applications qui partagent une partie de leur contexte ?

**Performances :** Le code d'observation est-il assez performant pour ne pas perturber les fonctionnement normal du logiciel ?

Enfin, concernant les stratégies d'adaptation du logiciel :

**Analysabilité :** La nature et la forme des stratégies d'adaptation permet-elle d'effectuer des analyses statiques afin de garantir un comportement approprié ?

**Généricité et réutilisabilité :** Est-il possible d'écrire une stratégie d'adaptation qui puisse être réutilisée facilement dans plusieurs applications similaires ?

**Dynamicité :** Est-il possible de faire évoluer la stratégie d'adaptation en cours d'exécution afin de mettre en œuvre des scénarios d'adaptation non anticipés lors du développement ?

Nous avons séparés les différents travaux étudiés en trois catégories : tout d'abord ceux basés sur la notion d'intergiciel (Section 3.2), qu'il s'agisse de plate-formes existantes ou de nouvelles propositions ; puis, les systèmes basés sur des modèles de composants (Section 3.3), et enfin une dernière section regroupant divers travaux qui n'entrent pas dans ces deux catégories (Section 3.4). Enfin, nous concluons ce chapitre en présentant une synthèse des évaluations et en identifiant les différentes alternatives stratégiques possibles et leurs conséquences, avant d'indiquer celles que nous avons choisies pour notre propre proposition (Section 3.5).

## 3.2 Middlewares réflexifs et adaptatifs

### 3.2.1 Open-ORB

#### Description

**Introduction.** Open-ORB est un projet de l'équipe de Gordon Blair à l'Université de Lancaster visant à définir une architecture ouverte et réflexive pour le middleware [Blair and Coulson, 1997; Andersen et al., 2000; Blair et al., 2000]. L'idée de base est d'utiliser des techniques réflexives pour rendre le middleware plus ouvert et adaptable, en donnant accès aux mécanismes d'exécution de la plate-forme à travers un protocole à méta-objets [Kiczales et al., 1991].

**Composants.** Open-ORB est basé sur un modèle de composants inspiré du modèle de référence RM-ODP [Blair and Stefani, 1998] dont les caractéristiques sont :

– la possibilité pour un composant d'offrir plusieurs interfaces, et de déclarer des dépendances envers d'autres interfaces ;

- le support des interfaces pour média continus (flots de données), en plus des interfaces traditionnelles basées sur l'envoi de message ;
- la gestion explicite des interactions entre composants (à travers des objets de liaison (*binding objects*), qui rend explicite l'architecture de l'application ;
- et le support en standard d'un service de notification d'événements.

**Méta-niveau.** Ce modèle de base est complété dans Open-ORB par un niveau méta appelé *meta-spaces* qui associe à chaque composant quatre modèles (*meta-space models*, cf. figure 3.1) :

- le méta-modèle de *composition*, qui permet d'accéder au graphe d'objets constituant un composant et de le modifier ;
- le méta-modèle d'*encapsulation*, qui donne accès à l'interface d'un composant à travers la liste des méthodes qu'il supporte et des attributs qu'il définit ;
- le méta-modèle de *ressources*, permettant de connaître l'utilisation des ressources système (mémoire, *threads*...) et dans une certaine mesure de les influencer (modification de l'algorithme d'ordonnancement des *threads* par exemple) ;
- et le méta-modèle d'*environnement*, qui représente l'environnement d'exécution de chaque interface du composant, et qui réifie des fonctionnalités comme la réception d'un message, sa sélection...

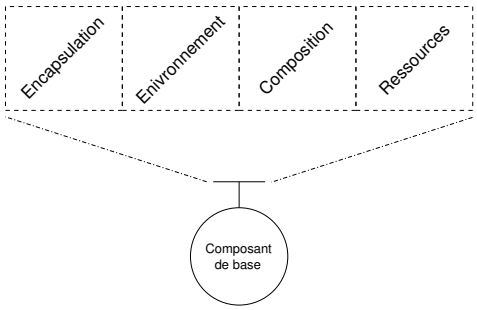


FIG. 3.1 – Structure d'un composant Open-ORB

**Tour réflexive.** Les éléments du *meta-space* étant implémentés avec les mêmes techniques que les objets de base, ils peuvent eux aussi avoir leur niveau méta. Cela introduit une récursion qui en théorie conduit à la construction d'une *tour réflexive* infinie. En pratique, les différents niveaux méta ne sont instanciés concrètement que s'ils sont nécessaires (instanciation paresseuse).

**Support de l'adaptation.** Open-ORB est conçu dans le but de permettre des adaptations relatives à la qualité de service (QoS, *Quality of Service*). Cette adaptation est réalisée en introduisant au niveau méta des composants spécifiquement chargés de gérer ces aspects. Pour que leur introduction se fasse de façon non-invasive, leurs interactions avec les composants de base se font uniquement par l'intermédiaire du modèle événementiel supporté en standard par le modèle de composants choisi. Ces *management components* sont ainsi notifiés automatiquement de ce qui se passe au niveau de base. On distingue deux types de *management components* : les *moniteurs* et les *contrôleurs*. Les moniteurs ont un rôle passif, ils se contentent d'observer le système et de recueillir des informations et des statistiques. Les contrôleurs quant à eux ont un rôle plus actif. Certains d'entre eux utilisent les informations récupérées par les moniteurs pour sélectionner une stratégie d'adaptation, tandis que les autres, appelés *activateurs* sont chargés d'implémenter la stratégie choisie. Par exemple, si un moniteur détecte une dégradation brutale de la qualité d'une connexion réseau, un sélecteur pourra choisir parmi les différentes stratégies à sa disposition la plus adaptée aux circonstances, par exemple supprimer le son d'une vidéo en cours de visionnage. Cette stratégie sera elle-même implémentée par un composant activateur.

**Dynamicité.** Le modèle Open-ORB est complètement dynamique, les différents composants existant à l'exécution. On peut donc ajouter, retirer ou reconfigurer ces composants à n'importe quel moment, y compris les composants du niveau méta. Cela signifie que les politiques d'adaptation elles-mêmes sont modifiables dynamiquement.

#### Évaluation

Open-ORB est un des rares systèmes à prendre en compte tous les aspects de l'adaptabilité dynamique, de la détection des changements de l'environnement à la modification incrémentale du système.

Les reconfigurations sont spécifiées par des méta-programmes généraux qui ont accès à toute la puissance de la réflexion pour modifier la structure et le comportement des composants, ce qui empêche d'effectuer des **analyses** pour vérifier par exemple que la cohérence du système est conservée. Le méta-modèle décrivant les composants est assez complet, ce qui permet des reconfigurations assez fines, et offre donc un bon niveau de **modularité**. En revanche, le niveau réflexif élevé (tour réflexive) sans optimisations particulière (en tout cas aucune n'est décrite) implique probablement des **performances** assez faibles. Globalement, Open-ORB a tous les avantages et tous les inconvénients des approches réflexives.

Concernant l'observation du contexte d'exécution, si Open-ORB prévoit explicitement la présence de *moniteurs*, aucun support n'est fourni pour leur création ou leur gestion. La plupart de nos critères ne sont donc pas applicables. Cependant, notons que cette approche a l'avantage de permettre à Open-ORB d'utiliser facilement des solutions de monitoring existantes.

La situation concernant les stratégies d'adaptations est similaire : leur présence est prise en compte sous la forme des *contrôleurs*, mais il s'agit de programmes **génériques**, et donc pas **analysables**.

### 3.2.2 QuO

#### Description

**Introduction.** Le système QuO (*Quality Object*) [Zinky et al., 2002; Pal et al., 2000; Loyall et al., 1998] est une plate-forme intergicielle qui étend la norme CORBA[1] en mettant l'accent sur la gestion de la Qualité de Service et l'adaptation des applications. L'approche proposée introduit un nouveau rôle dans le cycle de développement des applications, celui de concepteur QoS, chargé spécifiquement de la conception des stratégies d'adaptation ; QuO propose un ensemble de technologies qui étendent CORBA pour supporter ce nouveau rôle.

**Mesure des composants.** Les mesures des performances réelles des applications sont effectuées par des sondes introduites dans les objets CORBA applicatifs. QuO utilise pour cela un langage spécifique nommé ASL et des techniques type AOP pour tisser le code de mesure dans le code applicatif. Les résultats obtenus pour chaque objet sont corrélés pour obtenir des mesures globales de la QoS effective de l'application.

**Mesure des ressources.** Les mesures concernant le niveau d'utilisation des ressources sont effectuées par des *System condition objects* capables d'observer (et parfois de manipuler) les ressources systèmes et les mécanismes propres à l'intergiciel. QuO ne fournit pas d'implémentation réelle de ces objets pour mesurer les ressources systèmes, mais est capable de communiquer avec plusieurs solutions de monitoring existantes.

**Contrats de QoS.** Une fois obtenues des mesures concernant d'une part le fonctionnement des objets et d'autre part l'utilisation des ressources système, un *contrat de QoS* est utilisé pour déterminer, le cas échéant, les réactions nécessaires pour adapter le fonctionnement de l'application. Ce contrat est écrit dans un langage spécifique, CDL[2], qui définit :

[1] Common Object Request Broker Architecture
[2] Contract Definition Language

- des *régions opérationnelles* indiquant dans quel état se trouve l'application en fonction des valeurs de différentes mesures obtenues ;
- les *réactions* optionnelles à appliquer lors d'une transition entre deux régions opérationnelles. Ces réactions consistent uniquement à envoyer un événement à l'application, qui est libre d'y réagir comme elle le veut.

**« QoS Delegates ».** Les *QoS delegates* sont des objets CORBA qui encapsulent les objets applicatifs afin de les rendre adaptatifs. Ils modifient le flot d'exécution des objets en interceptant les envois de message afin de pouvoir éventuellement en modifier la sémantique. Lorsque l'objet encapsulé envoie un message, le délégué l'intercepte et interroge le contrat de QoS pour connaître l'état courant de l'application, modélisée par les régions opérationnelles. Le délégué peut alors choisir, en fonction de ces informations, d'interpréter le message d'un façon appropriée, adaptant ainsi le comportement de l'objet au contexte d'exécution courant, tel que modélisé par le contrat.

#### Évaluation

QuO supporte deux types de « reconfigurations ». Le premier, implémenté dans les *QoS Delegates* et similaire à celui supporté par DART (cf. Section 3.4.2), consiste à modifier dynamiquement le flot d'exécution des messages en fonction de l'état courant du contexte. Le second est déclenché par des notifications d'événements envoyés au code applicatif, qui peut alors réagir comme bon lui semble. Dans les deux cas, le comportement résultant ne peut pas être **analysé** statiquement et il est impossible de garantir la consistance de l'application. Le reconfiguration sont **modulaires**, puisqu'elle peuvent cibler des objets, voire des méthodes individuelles. Il est impossible d'évaluer les **performances** des reconfigurations elles-mêmes, puisque celles-ci sont arbitraires. Les reconfigurations effectuées par les délégués sont programmées au niveau méta, et donc génériques et **transparentes** pour l'application, mais elles ne semblent pas pouvoir être spécifiées ou modifiées dynamiquement.

Comme son nom l'indique, le système est très orienté vers l'optimisation des performances. Les seuls éléments du contexte pris en compte concernent donc l'utilisation des ressources système (matériel essentiellement) et, de façon plus originale, le niveau d'utilisation des services fournis par la plate-forme. L'implémentation réelle de l'observation est délégué à des systèmes existants, donc il n'est pas possible d'appliquer ici nos critères d'évaluation. Concernant les données manipulées dans les contrats, elles sont peu structurées entre elles, mais sont **enrichies** par des méta-données décrivant par exemple le niveau de confiance que l'on peut accorder à leur valeur.

Les stratégies sont exprimées séparément du code applicatif, dans un langage dédié (DSL). La forme de ce langage devrait permettre d'effectuer des **analyses** de leur comportement, mais les auteurs n'en mentionnent aucune. Ces « contrats de QoS » sont peu génériques, puisqu'ils référencent explicitement des objets métiers afin de spécifier les notifications d'événements lors d'un changement de région opérationnelle. Ces stratégies sont transformées à la compilation dans un langage cible (C++ ou Java) puis intégrées à l'application, et ne peuvent donc pas être modifiées **dynamiquement**.

### 3.2.3 dynamicTAO

#### Description

**Introduction.** dynamicTAO est un ORB[3] réflexif conforme au standard CORBA développé dans le cadre du projet 2K [Kon and Campbell, 1999; Kon et al., 2000; Romàn et al., 2001], et dont le but est de rendre les systèmes patrimoniaux (*legacy systems*) capables de s'adapter à la grande diversité des environnements d'exécution actuels.

**Reconfiguration dynamiques.** Comme son nom l'indique, dynamicTAO est une extension d'un ORB existant, TAO[4]. TAO est un ORB libre (au sens logiciel libre) écrit en C++ et qui a la particularité

[3]Object Request Broker
[4]The ACE ORB

d'être portable, extensible et surtout facilement configurable : les différentes parties du moteur de l'ORB sont implémentées sous la forme de modèles de conception *Stratégie*, ce qui permet de choisir parmi plusieurs implémentations la mieux *adaptée*. Cependant, dans l'implémentation originale, ce choix se fait au démarrage du système, en lisant un fichier de configuration. dynamicTAO ajoute la possibilité d'effectuer cette reconfiguration à la volée, pendant l'exécution du programme. Plus précisément, les opérations permises sont :

- migration de composants vers d'autres sites du système distribué ;
- chargement et déchargement de modules du middleware pendant son exécution ;
- inspection et modification de la configuration de l'ORB.

Ces opérations sont accessibles à travers un objet `DynamicConfigurator`, qui permet de découvrir l'ensemble des alternatives possibles pour un service donné (gestion de la concurrence par exemple) et d'activer ou désactiver une implémentation particulière de ce service. Un système de gestion de dépendances sophistiqué, géré par des *configurateurs de composants*, permet de conserver la cohérence du système lorsque le système est ainsi modifié.

**Observation du contexte.** Deux des services les plus intéressants sont le *2K Monitoring Service*, capable de recueillir des informations sur les interactions (envois de messages) entre les composants du système, et le *2K Resource Manager*, qui permet de connaître l'état d'utilisation des ressources physiques du système. En couplant ces deux services, on peut obtenir une connaissance très complète du système, nécessaire pour décider des modifications à effectuer et du moment où celles-ci doivent être effectuées.

### Évaluation

Les reconfigurations supportées par dynamicTAO correspondent au choix de l'implémentation la plus appropriée pour chaque service fourni par le middleware parmi celles qui sont disponibles. S'il semble possible de créer de nouvelles implémentations de ces services, celles-ci doivent être connues statiquement. Les auteurs ne précisent pas s'il est possible de créer de nouveaux services, mais le choix de conformité à une norme comme CORBA limite de toute façon les possibilités d'**ouverture**. Quels que soient les choix effectués, dynamicTAO **garantit** que tous les services requis par les différents composants sont présents, et mieux, grâce au `DynamicConfigurator`, que les choix de leurs différentes implémentations sont compatibles entre elles (les auteurs ne donnent aucun détail concernant les critères de compatibilité). Comme pour beaucoup de middleware adaptatifs, il n'est pas possible d'effectuer les reconfigurations de façon **modulaire** : tous les choix sont globaux et affectent automatiquement tous les composants applicatifs de la même manière. Si la plate-forme elle-même est très performante (TAO, le système d'origine, est orienté temps réel dur et destiné à des applications militaires, comme l'avionique par exemple), on ne peut rien dire sur les **performances** des reconfigurations elles-mêmes, les algorithmes et protocoles utilisés n'étant pas décrits. En ce qui concerne la **transparence** des reconfigurations du point de vue des programmeurs d'applications, on retrouve les avantages et inconvénients de CORBA : la standardisation (de l'IDL[5] entre autre), et lourdeur (relative) du processus de développement.

La **richesse** des informations sur le contexte est relativement limitée puisque qu'elle ne concerne que les ressources physiques et ne peut pas être **ouvert** et **généralisé** à d'autres types d'information. En contrepartie, le système est capable de prendre en compte les interactions entre composants, ce que peu de systèmes concurrent supportent.

La plus grosse faiblesse de dynamicTAO est qu'aucun support n'est proposé pour la spécification des stratégies d'adaptation, dont la programmation est intégralement à la charge des développeurs d'applications.

[5] Interface Definition Language

### 3.2.4 Middleware Control Framework

#### Description

**Introduction.** [Li and Nahrstedt, 1999] présente un framework pour la construction d'intergiciels supportant les adaptations de type Qualité de Service ($QoS$). Le cadre proposé est très inspiré des travaux sur la théorie du contrôle et les systèmes asservis, ici transposés dans le monde du logiciel.

**Niveaux d'abstraction.** L'une des caractéristiques fondamentales du système proposé est la distinction très forte qu'il fait entre le niveau « système » et celui des applications : le premier est caractérisé par un niveau d'abstraction très faible (protocoles réseau par exemple) et par des préoccupations globales (systémiques) comme l'équité (*fairness*) du partage des ressources entre applications ; le second niveau, applicatif, se préoccupe de notions plus abstraites, comme le nombre d'images traitées par secondes, et surtout spécifiques à une application. Là où la plupart des approches classiques de la gestion de la qualité de service se préoccupent surtout du premier niveau, [Li and Nahrstedt, 1999] proposent une approche dénommée « *application-aware QoS adaptation* », qui prend en compte les deux niveaux.

**Architecture.** L'architecture proposée reflète la distinction ente les deux niveaux d'abstraction. On retrouve donc deux sous-frameworks, l'un pour gérer les aspects globaux, et l'autre les aspects spécifiques aux applications. Puisque les auteurs considèrent que le processus d'adaptation du logiciel correspond naturellement à celui des systèmes en général, tels qu'étudiés dans le théorie du contrôle, le fonctionnement de chacun de ces sous-frameworks est basé sur des résultats de cette théorie. Le premier framework est appelé *Task Control Framework*, et est chargé de prendre les décisions d'adaptation globales ; le second, qui fait correspondre ces décisions aux besoins spécifiques de chaque application, est appelé *Fuzzy Control Framework*. Ils sont représentés au niveau du logiciel par deux « composants », respectivement l'*Adaptor* et le *Configurator*.

**Task Control Framework.** Ce premier framework est donc chargé de prendre des décisions d'adaptations qui s'appliqueront à tout le système, avec l'objectif de pouvoir ainsi garantir certaines propriétés globales comme l'équité du partage des ressources ou la stabilité du système. Pour cela, ce framework réalise un contrôle *actif*, c'est-à-dire que les décisions qu'il prend doivent obligatoirement être appliquées par les applications. L'intérêt de cette approche est qu'elle permet d'effectuer des vérifications formelles sur le comportement de ce framework, et donc du système, tout en séparant la prise de décision des détails des reconfigurations spécifiques aux applications. Chaque ressource prise en compte dans l'adaptation du système, comme par exemple le processeur ou le réseau, est représentée par un *Adaptor*s, lui-même constitué de deux parties : la tâche d'observation, chargée du *monitoring* de la ressource en question, et la tâche d'adaptation, qui prend les décisions. Les décisions d'adaptation prises à ce niveau concernent l'allocation des différentes ressources entre les applications qui s'exécutent dans le middleware.

**Fuzzy Control Framework.** Ce second framework, instancié dans chaque application, récupère les décisions globales de chaque *Adaptor*, et est chargé de les faire correspondre aux possibilités d'adaptation spécifiques à l'application. Le *Configurator*, qui implémente ce framework, applique pour cela un ensemble de règles de type **si ... alors**, qui à chaque niveau d'allocation de ressource indique l'adaptation correspondante à appliquer. Les types de reconfigurations supportés ne sont pas spécifiés puisque l'objectif du framework est justement de faire abstraction de ces détails.

#### Évaluation

Les possibilités de reconfiguration des applications ne sont pas spécifiées : les auteurs les considèrent en dehors du cadre de leur framework, qui traite essentiellement des stratégies d'allocation de ressources. Chaque application est responsable d'implémenter les mécanismes qui lui conviennent.

L'observation du contexte d'exécution est prise en compte par la tâche *monitoring* de l'*adaptor*, mais aucun détail n'est donné. Étant donné les types d'exemples décrits, on peut seulement supposer que les

auteurs ne considèrent sans doute que des critères de types ressources matérielles (processeur, mémoire, etc.).

Les stratégies d'adaptation sont séparées en deux niveaux. Au niveau global (systémique), le « *Task Control Framework* » dispose d'un *adaptor* pour chacune des ressources système observées, avec un algorithme d'allocation fixe et spécifique de cette ressource particulière (cette utilisation d'algorithmes spécifiques pour chaque ressource est à rapprocher de la notion de planificateur dans QuA, cf. Section 3.2.7). Ces algorithmes sont basés sur des résultats formels de la théorie du contrôle, dont les propriétés (stabilité et agilité en particulier) sont **prouvées** dans l'article. Cette partie du framework est complètement **générique**, puisqu'explicitement conçue pour être indépendante des applications. En revanche, elle est totalement **figée**. Au niveau local, les stratégies spécifiques à chaque application, sont représentées par de simples règles *si/alors*, ce qui les rend *a priori* facilement analysables, mais les actions correspondantes (dans les parties ALORS) sont complètement arbitraires, et il n'est donc **pas possible de raisonner** sur leur comportement.

### 3.2.5 Extension de ScalAgent

#### Description

**Introduction.** ScalAgent est un intergiciel orienté messages (*MOM : Message-Oriented Middleware*). [Quema and Bellissard, 2002] présente une extension de ce système permettant d'optimiser la configuration du middleware en fonction des besoins non-fonctionnels exprimés par les applications.

**ScalAgent.** Le middleware ScalAgent fournit un modèle de communication asynchrone inspiré des acteurs [Agha, 1985]. Il est constitué d' un canal de communication (« *channel* »), qui assure le transport des messages, et d'un « moteur » (*engine*) qui s'occupe de router les messages reçus vers les composants appropriés. Le middleware lui-même est construit à base de composants et des propriétés non-fonctionnelles peuvent y être ajoutées, soit dans le moteur (cas de la persistance par exemple), soit dans le canal de communication (par exemple pour assurer le sécurité des communications). Chaque composant non-fonctionnel peut ajouter des attributs aux composants applicatifs pour réaliser sa fonction ; par exemple, le composant qui implémente le service de persistance utilise un attribut `"location"` associé à chaque composant applicatif persistant pour savoir où le sauvegarder. Dans ce système, la construction d'un application se fait en trois phases :

1. *Description* de l'architecture de l'application grâce à un ADL [Medvidovic and Taylor, 2000] basé sur Olan [Balter et al., 1998] mais utilisant un syntaxe XML[6].
2. *Instanciation* de cette architecture, en spécifiant quels composants doivent être déployés sur quels sites.
3. *Déploiement* des composants sur les différents sites, sur chacun desquels doit se trouver un middleware ScalAgent compatible avec les besoins non-fonctionnels des composants.

**ADL étendu.** Bien que ScalAgent soit modulaire et configurable, dans la version standard du système, l'infrastructure est homogène : tous les sites disposent d'une configuration identique du middleware. Ceci peut poser de gros problèmes si certains des hôtes ont des capacités restreintes, car ils doivent quand même supporter l'ensemble des fonctionnalités utilisées par tous les composants de l'application. La première extension de ScalAgent proposée dans [Quema and Bellissard, 2002] concerne l'ADL utilisé pour décrire les composants applicatifs et l'architecture de l'application. Étant basé sur XML, cet ADL est facilement extensible. La version étendue de ce langage permet la spécification des propriétés non-fonctionnelles directement dans le langage en indiquant les valeurs des attributs gérant ces propriétés. Par exemple, la description XML d'un composant qui doit être persistant contiendra un élément supplémentaire indiquant comment et où il doit être sauvegardé. Pour chaque composant applicatif, il est aussi possible de spécifier

[6]Extensible Markup Language

l'ensemble des sites sur lesquels il peut être déployé ; si rien n'est spécifié, il pourra *a priori* être déployé sur n'importe lequel.

**Déploiement.** L'algorithme de configuration et de déploiement est étendu pour prendre en compte les nouvelles informations ajoutées à l'ADL. Il utilise une analyse de coûts pour déterminer pour chaque site la configuration optimale du middleware (en levant la contrainte d'homogénéité) et les composants applicatifs à y déployer. Ainsi, seuls les services réellement utiles aux composants présents sur un site y sont configurés. Pour évaluer les coûts, l'algorithme interroge chacun des composants fournissant les services non-fonctionnels en lui indiquant les caractéristiques matérielles du site, le nombre total de composants à y déployer, et le nombre de ces composants que le service doit prendre en charge. Le composant renvoie un entier correspondant à son évaluation du coût de la gestion de la propriété non-fonctionnelle dans ces circonstances. Cette analyse de coût, qui prend en compte, entre autre, les caractéristiques matérielles de l'hôte, est à rapprocher de nos notions de fonction d'adéquation et de contexte d'exécution. Ainsi, la nouvelle version de ScalAgent proposée est capable d'adapter la répartition des composants fonctionnels et des services non-fonctionnels en fonction de leur contexte, ici limité aux caractéristiques matérielles des hôtes disponibles.

### Évaluation

Ce système ne supporte que deux types d'opérations de déploiement : l'instanciation de certains services sur les noeuds du réseau, et la répartition des composants applicatifs sur ces mêmes noeuds. On peut difficilement parler de *re*configuration dans ce cas puisque ces deux choix sont fait statiquement, une fois pour toute, et le système ne semble pas supporter de modifications dynamiques (pour cette raison, parler de **performances** n'a pas vraiment de sens ici). Un des avantages de cette approche est que le système offre la **garantie** que chaque composant sera déployé sur un noeud qui fournit tous les services dont il a besoin. En revanche, la **modularité** est assez limitée, puisque les services sont spécifiés noeud par noeud (cette limite est inhérente à l'approche traditionnelle du middleware) : si deux composants se trouvent sur un même noeud, ils doivent partager la même implémentation des services qu'ils ont en commun. Le système semble assez **ouvert**, puisqu'il est possible d'ajouter de nouveaux services à la plate-forme et d'en faire bénéficier les composants applicatifs en ne modifiant que leur description dans l'ADL.

Au niveau de la connaissance du contexte d'exécution, les seuls éléments de « contexte » semblent être les différents services disponibles ou non sur les différents noeuds, et des caractéristiques non précisées desdits noeuds. Bien que cela n'ait pas trop de sens ici, on peut dire que d'une certaine façon, la **précision** des informations est parfaite, puisque l'on sait exactement quels services et quels composants sont présents sur chaque noeud. En terme de **richesse**, le système va plus loin que la seule présence ou absence d'un service grâce à l'analyse des coûts que doivent implémenter les services.

Concernant les stratégies d'adaptation, le système est **fermé** puisqu'il n'en supporte en fait qu'une seule, fixe, basée sur une évaluation des coûts au moment du déploiement (même si celle-ci est être influencée par les différents services). La solution proposée en donc plus proche de l'optimisation de déploiement que de l'adaptation, puisque les choix statiques ne sont jamais remis en cause. Malgré ce caractère statique, la possibilité d'ajouter de nouveaux services, qui influencent le choix de l'algorithme de déploiement, rend le système relativement **générique**. L'avantage d'une stratégie fixe, est qu'elle est *a priori* facilement **analysable** ; on pourrait même imaginer prouver l'optimalité de ses choix de déploiement étant données les coûts indiqués par les services (mais les auteurs ne mentionnent pas cette possibilité).

Globalement, la solution proposée ne nous semble pas assez dynamique et trop limitée au niveau des possibilités de configuration ; elle ne peut convenir qu'à des applications statiques s'exécutant sur des réseaux qui ne changent jamais.

### 3.2.6 CARISMA

#### Description

**Introduction.** CARISMA[7] [Capra et al., 2003] est une plate-forme intergicielle dédiée à l'exécution d'applications sur des hôtes mobiles. Alors que l'objectif traditionnel des intergiciels est la *transparence*, c'est-à-dire l'isolation des applications des détails de leur contexte, CARISMA est au contraire conçue pour permettre aux applications d'être conscientes de ce contexte afin de pouvoir s'y adapter, tout en leur simplifiant cette tâche au maximum.

**Architecture.** La plate-forme CARISMA est conçue en utilisant des techniques réflexives afin d'offrir le niveau de dynamicité et de contrôle nécessaire aux adaptations, mais la réification de la plate-forme visible des applications n'est pas directement basée sur un MOP comme dans [Ledoux, 1999] ou [Blair et al., 2000]. Le système est vu par les applications comme un fournisseur de *services abstraits*, comme par exemple l'envoi de message, chacun de ces services pouvant supporter plusieurs *politiques* (implémentations concrètes) différentes. Dans le cas de l'envoi de message par exemple, il peut exister une politique standard qui envoie les messages tels quels, une autre qui les compresse pour utiliser moins de bande passante, et une troisième qui les crypte avant de les envoyer. L'interface réflexive de la plate-forme permet aux applications de découvrir les différents services et politiques disponibles.

**Profils applicatifs.** Chaque application spécifie dans un *profil* la façon dont elle désire utiliser les services fournis par la plate-forme. Un tel profil indique, pour chaque service abstrait, les politiques choisies par l'application en fonction du contexte d'exécution dynamique. Par exemple, une application peut décider d'utiliser l'envoi de message standard s'il y a suffisamment de bande passante, et l'implémentation qui compresse les messages dans le cas contraire. Ce profil s'exprimerait de la façon suivante :

```
messagingService
  plainMsg
    bandwidth > 40%
  compressedMsg
    bandwidth < 40%
```

Dans ce cas, lorsque l'application invoque le service d'envoi de message de la plate-forme, celle-ci décide dynamiquement, en fonction du contexte d'exécution courant et du profil de l'application, quelle implémentation concrète utiliser. Les profils sont spécifiques à chaque application car seules les applications connaissent leurs priorités et peuvent décider ce qui est important pour elles.

**Conflits.** Lorsque plusieurs applications s'exécutent dans une instance de l'intergiciel, ou lorsque des applications distribuées sur plusieurs hôtes communiquent, les profils de ces applications peuvent entrer en conflit. CARISMA distingue deux types de conflits : *intra-profils* ou *inter-profils*. Les conflits intra-profils se produisent lorsque les règles qui constituent le profil d'une application donnée sont ambiguës et ne permettent pas de choisir l'implémentation du service à utiliser, ce qui peut se produire si les conditions associées à chaque politique ne sont pas exclusives. Le second type de conflits se produit lorsque plusieurs applications font des choix différents et incompatibles. Ces conflits peuvent se produire localement entre deux applications qui tournent dans une même instance de l'intergiciel[8] ou bien entre deux applications tournant sur deux hôtes différents mais qui dépendent l'une de l'autre.

**Résolution des conflits.** La détection et la résolution de ces conflits ne peut pas se faire statiquement, puisque dans le cas des conflits inter-profils, les différents profils ne sont connus qu'à l'exécution. Dans le cas des conflits intra-profils il est impossible de savoir si les différentes conditions associées aux politiques

[7] Context-Aware Reflective mIddleware System for Mobile Applications

[8] Cela est dû en partie à l'architecture monolithique de CARISMA qui ne peut pas utiliser plusieurs implémentations d'un même service à la fois.

dans un profil donné sont exclusives ou non dans le cas général. CARISMA choisit donc une approche dans laquelle ces conflits sont résolus dynamiquement au cas par cas, à chaque fois qu'ils se produisent. L'approche choisie est assez originale puisqu'elle est basée sur un modèle économique : les différentes applications sont considérées comme des *consommateurs* qui sont en concurrence pour obtenir les services fournis par le *producteur* (l'intergiciel). Le protocole de résolution des conflits est basé sur la notion de vente aux enchères, limitée à un tour pour éviter les surcoûts en terme de communications. Lorsqu'un conflit se produit, chaque application envoie au système son enchère, qui représente la quantité d'« argent » virtuel qu'elle est prête à payer pour obtenir le service. Le système évalue l'ensemble de ces enchères et choisit la politique qui maximise le nombre d'applications satisfaites ; il transfère ensuite l'« argent » virtuel des comptes de ces applications vers le sien.

**Allocation des quotas.** Au démarrage, chaque application se voit allouer un quota, déterminé par l'administrateur ou l'utilisateur, qui correspond à la quantité d'argent qu'elle peut dépenser pour enchérir. À intervalle régulier, l'intergiciel redistribue aux applications l'argent qu'il a obtenu pendant les dernières enchères. La quantité redistribuée à une application est proportionelle aux nombres d'interactions auxquelles elle a participé pendant cette période, et inversement proportionelle à la quantité d'argent qu'elle a misé. Ces critères permettent de favoriser les applications qui jouent selon les règles et de défavoriser celles qui surenchérissent trop souvent.

#### Évaluation

Les reconfigurations dans CARISMA se font au niveau des mécanismes de la plate-forme intergicielle ; chacun de ces services peut avoir plusieurs implémentations différentes, et il est possible de changer dynamiquement d'implémentation. Puisque tous les services sont disponibles à tout moment et que leurs implémentations sont connues statiquement, les reconfigurations ne modifient pas les applications elles-mêmes et leur consistance est donc **garantie**. En revanche, une instance de l'intergiciel ne peut utiliser qu'une implémentation d'un service donné à la fois, et les reconfigurations sont donc **globales** ; il n'est pas possible d'adapter de façon spécifique une application ou une partie d'une application. Au niveau des **performances**, la reconfiguration globale impose le surcoût de la détection et de la résolution des conflits, mais le protocole utilisé a été optimisé spécialement pour être le plus rapide possible.

L'observation du contexte est prise en compte, mais les auteurs ne donnent aucun détail sur cet aspect de leur système. Tout ce que l'on peut dire à la vue des exemples, est que le modèle du contexte d'exécution semble assez **pauvre** et très **spécifique** à la problématique de la mobilité.

Les stratégies d'adaptation des applications sont exprimées par de simples règles *si/alors*, associant un prédicat simple sur l'état du contexte à une implémentation d'un service de la plate-forme. Ces stratégies sont **génériques** puisqu'elles ne font référence à rien de spécifique à une application. Prises isolément, la stratégie de chaque application semble donc facilement analysable. Cependant, la limitation des reconfigurations globales et le besoin de gestion des conflits qui en découle imposent à chaque application d'embarquer du code spécifique et complexe pour participer aux enchères. Ce code fait lui aussi partie de la stratégie de l'application mais, étant totalement arbitraire, n'est **pas analysable**. Enfin, il ne semble pas possible de modifier **dynamiquement** la stratégie d'une application, que ce soit les règles *si/alors* ou le code de participation aux enchères.

### 3.2.7 QuA

#### Description

**Introduction.** Le projet QuA [Amundsen et al., 2004; Staehli et al., 2004] propose une plate-forme intergicielle qui automatise le déploiement et l'adaptation dynamique des composants applicatifs.

**Principe de fonctionnement.** L'approche met l'accent sur la séparation entre d'une part les types de composants, et d'autre part leur(s) implémentation(s). Lors de la création d'une application, son concepteur se contente de spécifier les types des composants qui doivent être présents à l'exécution et

leurs interconnexions, mais ne précise pas quelle implémentation utiliser. En contrepartie, le concepteur doit indiquer pour chaque composant les critères de Qualité de Service minimaux que doivent remplir les implémentations compatibles. On obtient ainsi des types de composants étendus, qui en plus des traditionnelles signatures de méthodes, incluent des critères de QoS. Différentes implémentations peuvent être développées séparément, chacune faisant des choix et des compromis différents en terme de ressources utilisées et de performances, qui doivent elles aussi être précisées dans la description de l'implémentation. La découverte, l'instanciation et la configuration des implémentations appropriées, y compris en terme de QoS, sont entièrement pris charge par la plate-forme QuA.

**Protocole d'instanciation.** Lorsqu'un composant client qui s'exécute dans la plate-forme QuA requiert l'instanciation d'un autre composant, il doit passer par l'intermédiaire d'un service spécial, appelé QuAMOP, et lui indiquer le type du composant voulu et les critères de QoS minimaux requis. En fonction de la nature des critères de QoS utilisés, le QuAMOP délègue cette requête à un *planificateur d'implémentation* spécialisé. Celui-ci est chargé d'identifier, parmi toutes les implémentations concrètes de composants disponibles, la mieux adaptée aux critères indiqués. Le QuAMOP n'a plus qu'à instancier le composant sélectionné et à le renvoyer au client.

**Planificateurs.** Il existe de nombreux types de critères de QoS, qui reposent sur des modèles parfois très différents (ressources partageables ou non par exemple) et qui requièrent des algorithmes spécifiques pour déterminer si une implémentation est compatible avec les critères attendus. Il n'est pas possible de concevoir une plate-forme qui intégrerait tous les types possibles. QuA est capable d'intégrer facilement tout nouveau type de critère en ajoutant simplement un nouveau planificateur d'implémentation approprié.

**Adaptation dynamique.** Le fonctionnement de QuA tel que nous l'avons décrit jusqu'ici est capable de choisir l'implémentation *initiale* d'un composant la plus adaptée. Afin de supporter l'adaptation dynamique, QuA intègre des moniteurs qui observent les ressources utilisées par les composants. Si ces ressources évoluent au point de remettre en cause les choix qui ont conduit à l'utilisation d'une implémentation donnée, le système réagit en instanciant un nouveau composant du même type, mais adapté aux nouvelles circonstances, et en remplaçant l'ancienne implémentation par la nouvelle.

### Évaluation

Les seules reconfigurations dynamiques supportées par la plate-forme QuA consistent à remplacer intégralement l'implémentation d'un composant par une autre. De ce point de vue, le système est **fermé**, puisque l'approche choisie ne permet pas d'ajouter d'autres mécanismes de reconfiguration. Puisque toutes les implémentations disponibles d'un type de composant doivent être compatibles en terme de signature, ce mécanisme **garantit** la consistance de l'application, au moins du point de vue du typage. Cependant, le remplacement à chaud de composant peut poser d'autres problèmes de cohérence (transfert d'état, messages perdus...) qui ne sont pas évoqués par les auteurs. Ces reconfigurations sont **modulaires** puisqu'elles s'effectuent composant par composant.

Concernant l'observation du contexte d'exécution, la présence de moniteurs pour permettre l'adaptation dynamique est évoquée par les auteurs, mais sans donner le moindre détail.

La stratégie d'adaptation repose sur un **algorithme codé en dur**, qui consiste à choisir l'implémentation qui offre les meilleurs critères de QoS. Cependant, cet algorithme est guidé par les méta-données ajoutées aux types et aux implémentations de composants, et influencée par des planificateurs spécifiques à un domaine de QoS. Bien que l'algorithme général soit **analysable** (par exemple pour prouver l'optimalité de ses choix), la possibilité d'ajouter des planificateurs arbitraires réduit la portée de ces analyses.

## 3.3 Modèles de composants adaptatifs

### 3.3.1 Adaptive Components

#### Description

**Introduction.** Le modèle des « adaptive components », ou composants adaptatifs décrits dans [Boinot et al., 2000; Boinot, 2002] vise à faciliter l'écriture de composants Java capables de fonctionner de façon différente en fonction du contexte d'exécution dynamique, autrement dit des composants adaptatifs.

***Adaptation classes.*** Le système proposé est une extension du langage Java (un compilateur *ad hoc* est fourni qui génère du *bytecode* Java standard) permettant d'écrire des « classes d'adaptation » (*adaptation classes*). Une telle classe se distingue d'une classe Java standard en ce qu'elle fournit plusieurs implémentations différentes d'une même interface, chacune adaptée à des conditions d'exécution particulières, et définit de façon déclarative quelle implémentation doit être utilisée dans quelles conditions d'exécution. Ces conditions d'exécution sont supposées être réifiées par d'autres composants. L'exemple suivant (figure 3.2, tiré de [Boinot et al., 2000]) présente un composant capable de changer dynamiquement d'algorithme de compression de son (LPC[9] ou GSM[10], implémentés dans d'autres classes) en fonction de la quantité de bande passante disponible (réifiée par la classe `RtcpController`). Lorsque la bande passante est en dessous de 13,3 kbit/s, le composant `SoundEncoder` utilisera l'implémentation définie dans la classe `Lpc`, et dans le cas contraire il utilisera la classe `Gsm`. Bien entendu, ce changement d'implémentation se fait dynamiquement pendant l'exécution du programme.

```
adaptclass SoundEncoder adapts Encoder {
  RtcpController rtpControl;
  SoundEncoder(RtcpController _rtpControl) {
    rtpControl = _rtpControl;
  }
  when (rtpControl.bandwidth < 13.3) Lpc();
  when (rtpControl.bandwidth >= 13.3) Gsm();
}
```

FIG. 3.2 – Exemple de composant adaptatif

**Avantages.** Il serait bien sûr possible d'obtenir le même résultat « à la main », en implémentant une combinaison des modèles de conception *Observateur* et *Stratégie* [Gamma et al., 1994] (c'est d'ailleurs de cette façon que le compilateur implémente ces classes adaptatives en pratique), mais au prix d'un effort considérable en terme de programmation. De plus, le résultat serait difficile à comprendre et à faire évoluer, sans compter le risque beaucoup plus important d'introduire des erreurs.

#### Évaluation

L'**unique reconfiguration** supportée par ce système correspond au changement dynamique d'implémentation d'un composant, parmi plusieurs disponibles. Cette opération **garantit** que tous les composants conservent la même interface et donc que la consistance globale du système est conservée. Le transfert d'état nécessaire lors d'une reconfiguration se fait automatiquement, par simple copie. Si cela simplifie la tâche du programmeur de composant, cela implique que toutes les implémentations d'un composant donné doivent partager la même représentation des données, et limite donc les possibilités d'adaptation.

[9] Linear Predictive Coding
[10] Global System for Mobile Communications

Les reconfigurations sont spécifiées **statiquement** dans la définition des classes de composants, ce qui implique que toutes les instances d'une classe de composant seront adaptées **de façon identique** et que les programmeurs de composants doivent eux-mêmes apprendre les constructions de langage ajoutées par ce système.

Le contexte est supposé réifié par des objets Java normaux. Le système est capable d'instrumenter ces objets par transformation de code source pour notifier les composants adaptatifs qui en dépendent lorsque leur état (variables d'instance) change. Aucun support n'est fourni pour l'écriture et la gestion de ces objets réifiant le contexte.

Les stratégies d'adaptation sont spécifiées par des règles à la sémantique **formalisée** qui permet de vérifier que tous les cas de figure sont pris en compte (complétude) qu'il n'y a pas d'ambiguïté entre les règles. Cependant, ces règles sont intégrées directement dans le code source des classes d'adaptation, et ne sont donc ni **génériques** ni **dynamiques**.

### 3.3.2 MOLèNE

#### Description

**Introduction.** MolèNE [André and Segarra, 2000] est un framework destiné à faciliter l'écriture d'applications adaptatives dans le contexte des systèmes mobiles et distribués à grande échelle. Pour cela, MolèNE utilise des techniques réflexives pour séparer le code fonctionnel des fonctionnalités réalisant l'adaptation [Malenfant et al., 2001].

**Architecture.** Le code d'adaptation, qui implémente les stratégies, est programmé au niveau méta dans un *Mobile Environment Management System* (MEMS[11]), et le code fonctionnel correspond au niveau de base. Un MEMS est responsable d'adapter le comportement des composants du niveau de base en fonction de la disponibilité des ressources. Pour réaliser ceci, deux frameworks sont utilisés : DNF[12] et RF[13].

**Observation du contexte.** Le DNF est chargé de la surveillance du contexte d'exécution. Les informations qu'il obtient concernant l'environnement sont mises à la disposition du reste du système, qui peut aussi s'enregistrer auprès du DNF pour être notifié (de façon asynchrone) de l'occurrence de certains événements. Dans un environnement distribué, les DNFs des différents noeuds communiquent entre eux pour que chacun ait une vision globale du système. Dans un DNF donné, on distingue les moniteurs de bas niveau (BM[14]), qui acquièrent les données brutes (par exemple la valeur courante de la bande passante disponible), et les moniteurs de haut niveau (HLM[15]), qui synthétisent les informations fournies par d'autres moniteurs (BMs ou HLMs).

**Reconfigurations.** Le RF est la partie active de MolèNE. Elle permet de construire des composants adaptatifs, capables de changer d'implémentation en fonction des conditions d'exécution. Ces composants comprennent pour cela un `Controller`, à l'écoute des modifications de l'environnement (notifiées par le DNF), chargé de modifier la partie fonctionnelle du composant. Pour décider quelles modifications effectuer et dans quelles conditions, le contrôleur est configuré par un objet de type `AdaptationStrategy`, spécifié par le concepteur de l'application. Cette stratégie d'adaptation est en fait un *automate* dont les différents états représentent différentes conditions d'exécution, et dont les transitions correspondent à la réaction prévue lors du passage d'un état à un autre. Il y a deux types de réactions possibles :

- reconfiguration de la partie fonctionnelle du composant (par la spécification de paramètres) ;
- changement complet de l'implémentation du composant.

[11] Mobile Environment Management System
[12] Detection and Notification Framework
[13] Reactive Framework
[14] Base Monitor
[15] High-Level Monitor

**Protocole de reconfiguration.** MolèNE inclut un protocole permettant d'assurer que les modifications effectuées par le framework causent le minimum d'interférence avec le code fonctionnel. En particulier, dans le cas ou l'on doit changer dynamiquement l'implémentation d'un composant, un protocole à trois phases (désactivation, transition et activation) et l'utilisation du modèle de conception *Mémento* [Gamma et al., 1994] permet de s'assurer qu'aucune donnée fonctionnelle n'a été perdue.

La figure 3.3 présente la structure d'un composant MolèNE.

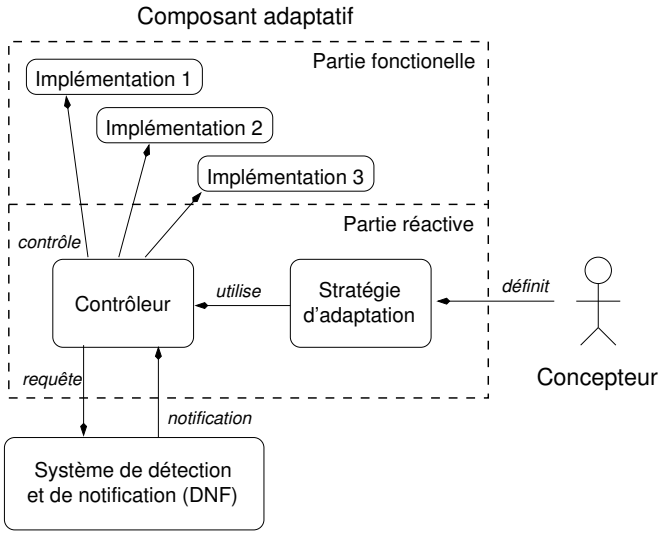


FIG. 3.3 – Structure d'un composant adaptatif dans MolèNE

### Évaluation

MolèNE supporte deux types de reconfigurations : paramétrage simple et changement complet d'implémentation d'un objet et ne permet pas d'en ajouter d'autres. Le système offre de bonnes **garanties** en utilisant un protocole de reconfiguration conçu spécialement. Chaque composant est adapté **individuellement**. Le transfert d'état qui s'effectue à chaque reconfiguration est basé sur le schéma de conception *Mémento* [Gamma et al., 1994] ; toutes les implémentations doivent donc se mettre d'accord sur le format de cet objet est doivent implémenter du code spécifique pour transférer leur état *de* et *vers* celui-ci.

Le système de surveillance du contexte d'exécution proposé par MolèNE, DNF, est l'un des plus **complets et sophistiqués** parmi ceux étudiés ici : il prend en compte la distribution des informations entre différents hôtes sur le réseau, et permet la définition de moniteurs dérivés. De plus, il est conçu pour être **générique** et extensible.

Les stratégies sont spécifiées sous forme d'automates, séparés du code de l'application et facilement **analysables**. Cependant, ces automates sont **spécifiques** à une classe de composants adaptatifs et l'association entre une classe et un automate se fait **statiquement**.

## 3.3.3 K-Components

### Description

**Introduction.** Le modèle des K-Components [Dowling et al., 2001; Dowling and Cahill, 2001] a pour objectif de permettre la création d'architectures logicielles dynamiques, condition nécessaire à la réalisation de logiciels adaptatifs. Pour cela, il est nécessaire d'aller au delà de l'approche de langages de description d'architecture, ou ADLs (*Architecture Description Language*), qui sont pour la plupart essentiellement statiques [Medvidovic and Taylor, 2000]. Le modèle proposé par Dowling repose sur la notion

de *réflexion architecturale* [Cazzola et al., 1998], qui applique les concepts de la réflexion non plus au niveau des constructions du langage de programmation, mais au niveau de l'architecture du logiciel : la structure du logiciel est réifiée et peut être manipulée à l'exécution pour modifier l'architecture du logiciel.

**Modèle architectural.** Dans le modèle proposé, l'architecture est considérée comme un graphe typé : chaque nœud du graphe représente une interface, ou port, et est annoté par l'identité du composant auquel elle appartient ; chaque arc représente un connecteur entre deux ports, et est annoté par les propriétés de la connexion. Puisque l'objectif est de permettre des reconfigurations dynamiques, ce graphe doit exister à l'exécution. Un composant spécial appelé *gestionnaire de configuration* (*configuration manager*) est chargé de maintenir cette réification du système et sert de point d'accès pour effectuer les reconfigurations.

**Reconfigurations.** Étant donné ce modèle, les reconfigurations dynamiques vont consister simplement en des transformations de graphe. Cependant, le type de reconfigurations supporté est limité. Afin de pouvoir garantir l'intégrité du système malgré son aspect dynamique, le modèle considère que la *structure* de graphe décrite plus haut est invariante et représente la structure statique du logiciel. Seules les *annotations* des nœuds et des arcs peuvent être modifiées. On peut donc reconfigurer les connecteurs eux-même en modifiant les propriétés de l'arc correspondant, ou bien modifier l'identité du composant associé à un nœud, et donc remplacer un composant par un autre dans une connexion. Le gestionnaire de reconfiguration, qui applique ces transformations au logiciel en cours d'exécution, utilise un protocole de reconfiguration qui garantit des propriétés transactionnelles aux reconfigurations (atomicité, cohérence du résultat, isolation...).

**Spécification des composants et de l'architecture.** Les K-Components sont définis en utilisant à la fois le langage IDL3 [Object Management Group, 2001] et C++. Dans un premier temps, IDL3 est utilisé pour spécifier l'interface des composants. Un gabarit C++ est ensuite généré à partir de cette interface, et le programmeur n'a plus qu'à le remplir avec le code du composant. La définition de l'architecture de l'application ne nécessite pas d'ADL, mais repose sur l'utilisation d'idiomes de programmation C++, qui sont analysés afin d'obtenir les informations nécessaires au gestionnaire de reconfigurations.

**Contrats d'adaptation.** Le code spécifiant quand et comment adapter les composants est spécifié dans des *contrats d'adaptation*, séparés du reste du code de l'application et exprimés dans un langage dédié. Ces contrats consistent en un ensemble de règles déclenchées par des événements dits *événements d'adaptation*. Lorsqu'un composant génère un tel événement (en utilisant les mécanismes d'événements du standard CORBA), les règles correspondantes sont exécutées par le gestionnaire de reconfiguration, ce qui conduit à une adaptation du système. Ces contrats sont définis statiquement et transformés en code C++ afin d'être compilés avec le reste de l'application.

### Évaluation

L'approche générale du modèle de K-Components, qui consiste à séparer clairement le code d'adaptation du reste de l'application est très proche de la nôtre. Cependant, un certain nombre des choix effectués, en particulier technologiques, ne permettent pas de tirer pleinement partie de cette approche.

L'approche utilisée pour effectuer les reconfigurations nécessaires à l'adaptation des applications remplit le critère de **garantie de consistance** des résultats en choisissant de ne supporter qu'un nombre restreint de mécanismes et en utilisant un protocole de reconfiguration qui assure des propriétés transactionnelles aux reconfigurations (sans dire exactement lesquelles). L'utilisation d'une approche réflexive pour représenter l'architecture permet d'effectuer n'importe quelle reconfiguration architecturale sans qu'elle n'ait été prévue explicitement par le programmeur, ce qui correspond à nos critères d'**ouverture** et de **transparence**.

Concernant la connaissance de son contexte par l'application, absolument *aucun support* n'est fourni par ce système. Il est à la charge du programmeur de chaque application d'implémenter de façon *ad hoc* le code d'observation nécessaire à son système et de générer les événements d'adaptation appropriés.

Les stratégies d'adaptation, spécifiées à base de règles réactives, remplissent les critères de **modularité** (le gestionnaire de reconfiguration supporte de multiples contrats d'adaptation), d'**analysabilité** (grâce à l'utilisation d'une forme très stricte, bien que la spécification des reconfigurations elle-même ne soit décrite nulle part par les auteurs), et d'agilité (de part leur nature réactive). Il semble par contre impossible de garantir la stabilité des stratégies, qui réagissent de façon automatique à chaque événement reçu. Si l'on veut éviter de trop nombreuses réactions inappropriées la seule façon est de filtrer les événements à la source, ce qui implique de rendre les composants qui les génèrent plus « intelligents » et qui va à l'encontre de la séparation de l'aspect d'adaptation. En ce qui concerne le critère de **généricité et de réutilisabilité** des stratégies (ici les contrats d'adaptation), il est difficile de se prononcer étant donné le peu de détails disponibles, mais étant donné le caractère réflexif du système, on peut supposer qu'ils sont remplis. Le critère de **dynamicité** quant à lui, bien qu'il ne soit pas rempli par l'implémentation actuelle, en grande partie parce qu'elle repose sur des technologies trop statiques, est parfaitement compatible avec l'approche proposée.

### 3.3.4 ACEEL

#### Description

**Introduction.** ACEEL[16] [Cherfour and André, 2003b,a, 2002] est constitué d'un modèle de composants et d'un *framework* Java permettant de créer des composants adaptatifs. Ce système a été conçu spécifiquement pour faciliter la création d'applications s'exécutant dans des environnements mobiles, où les ressources disponibles peuvent varier énormément au cours du temps.

**Structure des composants ACEEL.** Les composants ACEEL ont une structure inspirée du schéma de conception *Stratégie* [Gamma et al., 1994], qui permet de disposer de plusieurs implémentations alternatives pour une même interface, chacune adaptée à un contexte d'exécution particulier. L'interface d'un composant ACEEL est définie dans une classe abstraite, dont héritent toutes ses implémentations concrètes. Le schéma de conception *Stratégie* classique est semi-statique : le choix d'une implémentation parmi plusieurs, s'il peut se faire à l'exécution, se fait une fois pour toute. ACEEL étend ce schéma en permettant de changer dynamiquement l'implémentation utilisée par un composant, ce qui permet de toujours utiliser l'implémentation la mieux adaptée aux circonstances qui évoluent en cours d'exécution. De ce point de vue, ACEEL est donc en fait plus proche du schéma de conception *État*, qui permet de faire varier le comportement d'un objet en fonction de son état interne[17], mais étendu afin de prendre en compte non pas l'état interne de l'objet, mais son contexte d'exécution.

La partie fixe d'un composant, appelée « contexte », est la seule visible par les clients et implémente l'interface du composant en déléguant tous les appels vers l'implémentation concrète courante. L'une des particularités du modèle ACEEL – et sa limite principale – est que cette partie fixe encapsule aussi l'intégralité de l'*état* du composant, que les différentes implémentations concrètes doivent se partager. Les différents comportements d'un même composant doivent donc tous utiliser les mêmes données. Chaque composant ACEEL est géré au niveau méta par un *adaptateur*, qui décide quand changer l'implémentation du composant, en interprétant une *politique d'adaptation*. Notons aussi que bien qu'ACEEL soit qualifié de modèle de composant, il ne supporte en fait aucun moyen de structuration des applications, ni en terme de connexion entre « composants » ni en terme de composition hiérarchique (composites).

**Politiques d'adaptation.** Les politiques d'adaptation des composants sont spécifiées dans des scripts séparés du reste du code, et qui peuvent être chargés dynamiquement par les adaptateurs de composants. La syntaxe des politiques utilise XML. Chaque politique est constituée de deux parties. Dans la première est spécifié le cycle de vie de chaque comportement concret, c'est-à-dire quand le système doit créer et détruire une instance de chaque classe correspondant à un comportement concret : en même temps que la

[16] Adaptive ComponEnt modEL

[17] [Gamma et al., 1994] décrit l'objectif du schéma *État* ainsi : *Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.*

création / destruction du composant, ou bien seulement lorsque ce comportement est activé / désactivé. La seconde partie de la politique est la plus importante, elle spécifie les règles d'adaptation sous la forme de couples « *OnEvent* : *Action* ». *OnEvent* décrit un événement se produisant dans le contexte du composant qui déclenche la règle. *Action* décrit l'action d'adaptation à appliquer lorsque cela se produit. Les événements sont générés par un sous système de détection décrit plus loin. Les deux types d'action d'adaptation possible sont : paramétrage du composant, en modifiant la valeur d'un de ses attributs (seules les valeurs constantes semblent être supportées), ou bien choix d'un nouveau comportement du composant parmi les implémentations disponibles.

**Protocole de reconfiguration.** Afin d'effectuer les reconfigurations en toute sécurité, l'adaptateur associé à chaque composant ACEEL suit un protocole bien défini qui désactive le composant avant de le reconfigurer, puis le réactive en garantissant qu'aucun message destiné au composant n'a été perdu pendant sa reconfiguration.

**Contraintes pour les composants ACEEL.** Pour pouvoir garantir que le protocole ci-dessus fonctionne correctement, ACEEL impose des contraintes très fortes sur les composants :

- Tout d'abord, comme nous l'avons dit plus haut, les objets implémentant les différents comportements possibles d'un composant ne doivent pas avoir d'état propre. Ils doivent tous partager une même implémentation de l'état du composant, stockée dans la partie fixe de celui-ci. Cette contrainte existe pour permettre de changer facilement le comportement du composant sans avoir à résoudre le problème du transfert d'état, très complexe dans le cas général, et qui nécessite l'implémentation de $n \times (n-1)$ « convertisseurs » par le programmeurs dans le cas « manuel » (où $n$ est le nombre d'implémentations possibles du composant).
- Pour garantir que n'importe quelle implémentation du comportement puisse être choisie à n'importe quel moment, ACEEL impose que toutes les méthodes de l'interface du composant, qui sont donc implémentées par tous les comportements, aient *exactement le même effet sur l'état* du composant.

**Système de détection et notification.** Le framework ACEEL inclut un système d'observation de l'environnement d'exécution capable de notifier les adaptateurs de composants lorsque certains événements se produisent. Ce système est implémenté sous la forme d'un processus « démon » sur chacune des machines où s'exécutent des composants ACEEL. Ce démon se charge d'observer les ressources systèmes de l'hôte sur lequel il s'exécute, comme le réseau, la batterie ou la mémoire. Il communique avec les adaptateurs de composants à travers un protocole textuel implanté au dessus de TCP[18] et nommé ADMP[19]. Cette communication peut se faire selon deux modes : interrogation directe ou enregistrement / notification. Dans le premier mode, un adaptateur de composant peut demander au démon la valeur courante d'un attribut d'une ressource donnée (qu'il doit connaître à l'avance). Dans le second mode, l'adaptateur enregistre une condition de la forme $ressource.attributopseuil$, où $op$ est un opérateur de comparaison ($=, \geq, \geq, \leq, \leq$). Une fois cet enregistrement effectué, le démon notifiera l'adaptateur en question à chaque fois que cette condition deviendra vraie. Ainsi, lorsqu'une politique est chargée dans un adaptateur, il lui suffit de s'enregistrer auprès du système de notification pour chacune des conditions apparaissant dans les règles de la politique.

### Évaluation

L'unique mécanisme de reconfiguration supporté par ACEEL est au cœur du modèle et influence toute sa conception. Si cela permet au système d'offrir des **garanties** en ce qui concerne les reconfigurations (en particulier de consistance), ce choix a aussi des conséquences négatives. Tout d'abord, le modèle de composant *ad hoc* ne supporte ni les connexions entre composants, ni la composition hiérarchique. La granularité des adaptations est fixe, puisqu'il n'est possible d'adapter qu'un composant à la fois. Il

[18] Transmission Control Protocol
[19] ACEEL Distributed Monitoring Protocol

semble possible, par exemple en chargeant des politiques similaires sur plusieurs composants, d'adapter tout un groupe de composants, mais cela implique l'activation et le traitement de toutes les politiques individuellement, ce qui peut causer des problèmes de performances si l'on veut adapter beaucoup de composants. Cependant, les plus gros problèmes du modèle ACEEL concerne les critères d'**ouverture** et surtout de **transparence**. En effet, ACEEL impose aux programmeurs de composants des contraintes structurelles très fortes, en l'obligeant à utiliser un schéma de conception de type *Stratégie*. La séparation stricte imposée par ACEEL entre l'état des composants de leur comportement n'est pas naturelle pour les programmeurs. De plus, en imposant une même représentation des données et la même sémantique pour chaque implémentation alternative concernant ces données, le modèle limite les opportunités d'adaptation.

Le sous-système d'observation et de notification est lui beaucoup plus intéressant. Bien qu'il soit impossible d'évaluer la **précision** des informations, qui dépendent de l'implémentation des moniteurs de ressources, le système est conçu pour être très **général** et réutilisable. Étant donné la forme assez simple des expressions que le système est capable de détecter, on peut supposer que ses **performances** sont assez bonnes. Cependant, le défaut de ce système est son manque de **richesse** en terme de modélisation du contexte. Celui-ci est représenté par un ensemble de ressources décrites par des attributs, mais il n'y a aucun moyen de structuration entre ces ressources et la liste des moniteurs est statique et doit être connue pour pouvoir écrire des politiques. Il est donc impossible de réagir par exemple à l'apparition ou la disparition de ressources, à moins d'implémenter un moniteur spécifique pour cette condition.

Concernant les stratégies, leur expression sous forme de règles réactives à la forme simple permet d'envisager des **analyses formelles** de leur comportement – même si aucune n'est décrite explicitement – et garantit une certaine agilité, puisque leur interprétation est immédiate. Les politiques sont modulaires, et exprimées séparément du reste du code, ce qui permet une certaine **dynamicité** en autorisant l'ajout de nouvelles politiques en cours d'exécution. Cependant, cette ouverture est limité par la forme très contrainte des composants, puisque toutes les implémentations alternatives doivent être fournies à la compilation. La **généricité** des politiques semble très limitée, puisque le langage utilisé pour les exprimer oblige à désigner explicitement les ressources observées, les seuils et les actions d'adaptation sous forme de constantes ; une politique d'adaptation donnée n'est donc utilisable que pour un composant ACEEL spécifique.

Pour conclure, toutes les limitations du modèle ACEEL semblent découler d'une décision initiale qui considère l'adaptation comme étant la préoccupation principale. Comme le montrent bien les contraintes imposées aux programmeurs de composants ACEEL, le modèle est conçu explicitement pour faciliter le travail de l'adaptateur. L'utilisation du schéma de conception *Stratégie* force le programmeur d'application à décomposer son application en fonction de ce critère, au détriment des autres préoccupations. Le résultat fonctionne, et permet effectivement des adaptations intéressantes, mais à un prix pour le programmeur qui nous semble trop élevé.

### 3.3.5 PLASMA

#### Présentation

**Introduction.** PLASMA[20] [Layaïda and Hagimont, 2005a,b] est un modèle de composants conçu pour la construction de services multimédia adaptatifs. En effet, les applications multimédia étant gourmandes en ressources (traitement, affichage...) et sensibles aux variations de ces dernières (synchronisation des flux, perte de paquets réseau), il est particulièrement important pour elles de pouvoir s'adapter à leur contexte d'exécution. Pour cela, une application PLASMA est constituée de trois catégories de composants : les composants multimédia qui implémentent la partie *métier* de l'application, les composants d'*observation* chargés de détecter les changements du contexte d'exécution qui nécessitent une adaptation, et enfin les composants de reconfiguration, dits *actuateurs*, chargés d'effectuer les adaptations elles-mêmes en reconfigurant les composants métiers.

[20] PLAtform for Self-Adaptive Multimedia Applications

**Composants multimédia.** PLASMA utilise le modèle de composants Fractal [Bruneton et al., 2004] pour construire des composants de traitement multimédia, et les composer sous forme de « *pipelines* ». PLASMA distingue trois types de composants :

1. Les *Media Primitive*, composants primitifs qui implémentent une fonction multimédia bien définie, comme par exemple le décodage d'un format vidéo, ou bien l'application d'un filtre à un flux audio. Les interfaces entrantes et sortantes de ces composants sont typées par le format du flux qu'elles sont capables de gérer.
2. Les *Media Composites* représentent les fonctions de haut niveau de l'application multimédia, appelées *tâches*, comme l'encodage, le décodage et la transmission. Comme leur nom l'indique, ces *Media Composites* sont implémentés par composition de composants primitifs (*Media Primitive*) connectés par leurs interfaces. Cette composition repose sur les mécanismes du modèle Fractal sous-jacent, qui permet d'encapsuler un certain nombre de composants dans un composite et de n'exposer que certaines de leurs interfaces (entrantes ou sortantes) à l'extérieur.
3. Enfin, les composants *Media Session* représentent une application multimédia complète, constituée de plusieurs tâches implémentées par des *Media Composites*. Un composant *Session* fournit aussi à l'extérieur une interface qui permet de configurer et de contrôler l'application (démarrage, pause, arrêt...).

Une application multimédia construite avec PLASMA est donc constituée d'une ensemble de composants de plus ou moins haut niveau connectés les uns aux autres pour former une chaîne de traitement (*pipeline*) d'un ou plusieurs flux multimédia.

**Composants d'observation.** Pour ce qui est de l'observation du contexte de l'application, PLASMA distingue l'observation elle-même, implémentée par des composants *sondes*, de la détection des événements significatifs, implémentée par des *senseurs*. Deux types de sondes sont supportées :

1. les sondes *QoS*, qui sont implémentées par les composants multimédia eux-mêmes et fournissent des mesures de qualité comme par exemple le taux de perte de paquets pour un composant de transmission réseau ;
2. et les sondes *de ressources*, indépendantes d'une application particulière et qui observent les ressources système globales comme la mémoire disponible ou la charge d'une batterie.

Les senseurs quant à eux sont chargés de détecter les modifications significatives du contexte, et de générer les événements correspondant (qui permettront de déclencher une adaptation). Certains senseurs peuvent être liés à une sonde – QoS ou de ressource –, auquel cas ils génèrent un événement si la valeur observée dépasse un seuil donné. PLASMA supporte aussi des senseurs dits *externes*, qui permettent de détecter n'importe quelle condition qui n'entre pas dans le cadre de la QoS ou des ressources, mais nécessitent une implémentation spécifique.

**Actuateurs.** Les adaptations elles-mêmes, déclenchées par les événements générés par les senseurs, sont réalisées par des composants dits *actuateurs*. Ces composants primitifs peuvent reconfigurer n'importe quel composant multimédia par l'intermédiaire de l'interface de contrôle de ces derniers. Cette interface permet *(i)* de savoir si un composant dispose d'un attribut donné (`hasAttribute(name)`, *(ii)* d'obtenir la valeur courante d'un attribut (`getAttribute(name)`), et enfin *(iii)* de modifier la valeur d'un attribut (`setAttribute(name, value)`). Bien que cette interface de configuration soit la même pour tous les composants, chacun est libre d'interpréter la modification d'un attribut soit :

- comme la simple modification d'un paramètre de configuration, par exemple le taux de compression utilisé par un composant encodeur ;
- comme une reconfiguration structurelle, qui implique de modifier la façon dont les sous-composants d'un composite sont connectés, par exemple en choisissant un composant d'encodage (et donc un algorithme différent) ;
- ou bien encore comme une reconfiguration des sondes (par exemple la durée entre deux mesures), des senseurs (seuil de détection, ...) ou des actuateurs eux-mêmes.

**ADL dynamique.** PLASMA fournit un ADL basé sur XML pour décrire l'architecture d'une application multimédia adaptative. Cette description comprend à la fois les composants multimédia (tâches, composants primitifs et connexions entre eux), les différentes sondes à déployer, avec les seuils de détection, et enfin les actuateurs, qui sont entièrement décrits avec l'ADL sous la forme d'une suite d'actions de reconfiguration.

**Déploiement et exécution.** Étant donnée une description d'application, PLASMA instancie et connecte les différents composants (multimédia, sondes et senseurs, actuateurs) afin d'obtenir une configuration initiale. Pendant l'exécution, les sondes observent les critères de QoS et les ressources, et lorsqu'un senseur détecte le dépassement d'un seuil, l'événement qu'il génère est envoyé à l'actuateur correspondant. Ce dernier applique les différentes reconfigurations indiquées, adaptant ainsi l'application aux nouvelles conditions d'exécution.

#### Évaluation

Les reconfigurations permettant d'adapter une application PLASMA sont codées *statiquement* et intégrées directement dans la description de l'architecture de l'application. Elles ne sont donc *pas modulaires*. L'utilisation d'une interface uniforme à base d'attributs pour reconfigurer les composants simplifie l'écriture des actions de reconfiguration mais rend difficile de prévoir l'effet d'une modification de paramètre, puisque cette interface unique cache en fait plusieurs mécanismes de reconfiguration différents. Cette approche est relativement peu *transparente*, puisque les composants doivent implémenter explicitement une interface de reconfiguration. Une action de reconfiguration est une simple séquence de modification d'attributs, et d'après les exemples (qui utilisent tous des constantes pour les valeurs des attributs), on ne sait pas s'il est possible de référencer les valeurs de sondes pour déterminer la nouvelle valeur d'un attribut. En revanche, PLASMA *garantit* la consistance des reconfigurations grâce à l'utilisation d'un protocole qui évite la perte de messages au cours d'une adaptation.

Les mécanismes d'observation supportés par PLASMA couvrent tous les domaines nécessaires : l'application elle-même avec les sondes QoS, les ressource systèmes, et tout autre élément du contexte peut être observé par l'intermédiaire des senseurs externes. La *richesse* sémantique des informations obtenues par les sondes de QoS se fait au détriment de la *généricité* du mécanisme utilisé, puisque chaque composant doit intégrer (statiquement) le code d'observation. PLASMA ne permet pas de structurer les sondes, par exemple pour aggréger les valeurs de plusieurs d'entre elles, et le déclenchement d'événements se fait uniquement sur le passage d'un seuil constant. Cette relative simplicité des mécanismes utilisés garantie cependant de bonnes *performances*.

Les stratégies d'adaptation n'apparaissent pas explicitement, mais uniquement à travers les connexions effectuées entre sondes, senseurs et actions de reconfiguration. Sous cette forme (la définition utilisant l'ADL), elles sont probablement *analysables*. Les stratégies sont codées directement dans la définition de l'architecture, et donc ni *modulaires*, ni *génériques*, ni *dynamiques*. En revanche, elles sont bien intégrées dans le modèle de composant, sous la forme des composants sondes, senseurs et actions, et des connexions entre ces derniers. Enfin, PLASMA se limite à un domaine d'applications spécifique.

## 3.4 Autres approches

Cette section décrit d'autres approches pour la construction d'applications adaptatives, qui ne rentrent pas dans les deux sections précédentes.

### 3.4.1 Odyssey

#### Description

**Introduction.** Odyssey [Noble et al., 1997] est une extension du système d'exploitation NetBSD conçue pour permettre l'adaptation des applications aux environnements mobiles. L'idée au cœur du système

consiste à faire *collaborer* le système d'exploitation et les applications afin de réaliser des adaptations les plus efficaces possibles. En effet, si c'est le système d'exploitation qui contrôle les ressources – telles que les connexions réseau – permettant d'adapter le comportement applications, seules les applications savent quelles modifications sont appropriées pour leur cas particulier.

**Division des responsabilités.** Les auteurs identifient un continuum dans la manière de diviser les responsabilités liées à l'adaptation entre le système d'exploitation et les applications :

- d'un côté, le système se contente de proposer des opérations de reconfigurations aux applications, qui décident individuellement comment les utiliser pour s'adapter ;
- de l'autre, le système adapte l'ensemble des applications de façon transparente, et donc de façon identique.

Ces deux extrêmes posent problèmes : dans le premier cas, les adaptations des diverses applications ne peuvent pas être coordonnées ; dans le second, les adaptations ne peuvent être que très générales, et ne peuvent pas tirer partie de la sémantique spécifique des applications (par exemple, le système ne peut pas décider de lui-même si certains paquets réseau peuvent être agrégés pour envoyés en une seule fois ou s'ils doivent être envoyés immédiatement).

**Architecture.** Odyssey se présente comme un ensemble de gestionnaires de données (*Warden*) spécialisés dans un type de données (images, son...). Les applications délèguent la gestion de leurs données à ces gestionnaires, qui sont capables d'adapter intelligemment (puisqu'ils sont spécifiques à un type particulier de données) leur *fidélité* de représentation en fonction des ressources disponibles. Le noyau du système observe l'évolution de la disponibilité des ressources et prévient les applications lorsque ces disponibilités sortent du domaine de tolérance qu'elles ont spécifié. Les applications peuvent alors modifier leurs domaines de tolérances, et les gestionnaires sont ensuite chargés de modifier la fidélité des représentations des ressources pour rentrer dans le nouveau domaine de tolérance. L'ensemble des gestionnaires de données est coordonné par un composant central appelé *viceroy*.

**Reconfigurations.** Les reconfigurations supportées concernent exclusivement la fidélité des données manipulées par les applications. Chaque gestionnaire de données est ainsi capable de modifier la qualité des données afin d'ajuster la quantité de ressources utilisées. Par exemple, le gestionnaire spécialisé dans le type de données « image » peut utiliser un format comme le JPEG[21] dont le taux de compression est ajustable, afin d'utiliser plus ou moins d'espace disque ou mémoire.

**Processus d'adaptation.** Lorsqu'une adaptation enregistre un document dans un gestionnaire de données, elle spécifie une fourchette qu'elle considère acceptable pour la fidélité de ces données, et qui correspond à un *contrat* entre l'application et le gestionnaire. Le composant central d'Odyssey, le *viceroy*, est chargé d'observer les ressources disponibles sur le système et, lorsqu'elles ne sont plus suffisantes, demande aux gestionnaires de données d'ajuster la fidélité des données dont ils ont la responsabilité. Si cet ajustement fait sortir la fidélité d'un document de la fourchette spécifiée par son application d'origine, le *viceroy* prévient alors l'application de cette violation du contrat, et l'application peut négocier un nouveau contrat et/ou prendre les mesures nécessaires par rapport à la baisse de fidélité de ses données.

### Évaluation

En ce qui concerne les reconfigurations, les seules opérations supportées par Odyssey sont très spécifiques, puisqu'elles ne concernent que la fidélité de documents multimédia ; le système manque donc d'**ouverture**. Si cette approche limitée permet d'offrir des opérations spécifiques et donc *a priori* **performantes** (par exemple l'utilisation d'algorithmes de compressions dédiés au son ou à la vidéo), notons qu'une fois que l'on a diminué la fidélité d'un document, il est impossible de revenir en arrière et de récupérer une version plus fidèle, puisque des informations ont été perdues. Cette perte d'information que

[21] Joint Photographic Experts Group

les applications ne peuvent pas toujours limiter (puisque ce sont les *warden* qui prennent ces décisions) peut nuire au maintient de la *consistance* des applications. Même la **transparence** du système est relativement limitée puisque les applications doivent utiliser explicitement de nouveaux appels systèmes, et qu'aucun moyen ne semble être fourni pour préciser les fourchettes de fidélité acceptables autrement que directement dans le code de l'application.

Les auteurs donnent très peu d'informations concernant les données contextuelles utilisées par les *warden* et le *viceroy*. Étant donnée la nature limitée des reconfigurations prises en compte, ces données ne sont probablement pas très **riches** – mémoire et espace disque libre par exemple – et l'implémentation correspondante **non réutilisable**.

Le fourchettes de fidélité spécifiées par les applications pour leurs documents sont ce qu'Odyssey propose de plus proche d'une stratégie d'adaptation. Leur forme très simple, alliée à un ensemble fixé d'opérations de reconfiguration aux résultats relativement prévisibles permet sans doute au *viceroy* d'effectuer un certain nombre d'**analyses** afin de garantir un bon fonctionnement global du système et l'équilibrage du partage des ressources ; malheureusement, ces analyses ne sont pas décrites. Comme nous l'avons déjà remarqué plus haut, il ne semble pas possible de séparer la spécifications de ces « contrats » du code de l'application, ce qui en fait des stratégies **fermées** et empêche leur **réutilisation**, sauf à réutiliser le code de l'application lui-même.

### 3.4.2 DART

#### Description

**Introduction.** DART[22] [Raverdy and Lea, 1999] est une plate-forme de développement dont le but est de faciliter l'écriture d'applications distribuées. Par rapport à des systèmes comme CORBA, DART va plus loin en permettant l'adaptation dynamique des applications aux conditions d'exécution. Le système utilise pour cela une combinaison de techniques réflexives et de surveillance (*monitoring*) de l'environnement d'exécution.

**Reconfigurations.** Deux mécanismes de reconfiguration sont disponibles, tous deux basées sur des techniques réflexives : les méthodes adaptatives et les méthodes réflexives (resp. *adaptive methods* et *reflective methods*).

**Méthodes adaptatives.** Les méthodes adaptatives sont destinées à être utilisées au niveau de base, donc par le programmeur d'application. Chacune de ces méthodes peut disposer de plusieurs implémentations différentes, et chaque envoi de message va déclencher un processus de sélection pour déterminer laquelle de ces implémentations est la plus appropriée. Cette sélection est effectuée en fonction des conditions du moment. Ce mécanisme est assez proche d'un modèle de conception *Stratégie* [Gamma et al., 1994] dans lequel les stratégies concrètes correspondent à différentes implémentations de la méthode, excepté que le choix d'une stratégie particulière se fait dynamiquement et surtout automatiquement.

**Méthodes réflexives.** Le mécanisme des méthodes réflexives est très similaire, mais est destiné à être utilisé au niveau méta. La réception d'un message par un composant est interceptée par un *réflecteur*, chargé de le rediriger vers le ou les méta-objets appropriés. Le message est ensuite traité comme une méthode adaptative, mais au niveau du méta-objet. Se situant à un niveau d'abstraction plus élevé, les méthodes réflexives ont l'avantage d'être plus générales que les méthodes adaptatives (liées à une application particulière), et donc de pouvoir être réutilisés dans plusieurs applications.

**Stratégies.** La gestion de l'adaptation du système se fait grâce à des *politiques d'adaptation* associées aux composants, qui peuvent être définies soit par l'application, soit par les bibliothèques qu'elle utilise. Ces politiques sont notifiées par le système des changements dans l'environnement d'exécution ou dans les

[22] Distributed Adaptive Run-Time

préférences utilisateur, et peuvent donc y réagir. L'ensemble de politiques d'adaptation présentes dans le système est coordonné par le DART *manager*, afin d'éviter les adaptations incompatibles ou incohérentes entre elles. Pour faciliter cette coordination, les politiques sont organisées en trois niveaux d'abstraction (*système*, *middleware* et *application*) et, à chaque niveau, groupées en modules suivant leur domaine d'application (communication, *thread*...). Cette organisation permet d'ordonner les politiques : d'abord suivant leur niveau d'abstraction (les politiques les plus prioritaires étant celles de plus haut niveau d'abstraction), et ensuite en affectant des priorités aux modules d'un même niveau.

La figure 3.4 résume les différents composants du système DART et leur relations.

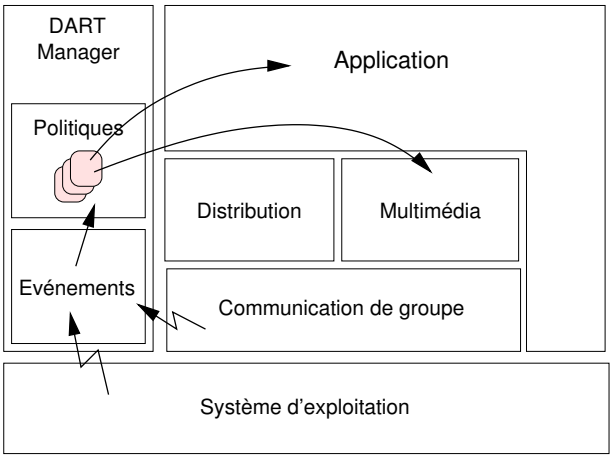


FIG. 3.4 – L'architecture de DART

Au niveau de l'implémentation, DART est basé sur une version modifiée de C++ (réalisée en utilisant OpenC++ [Chiba, 1995]) à laquelle de nouveaux mots-clef ont été ajoutés pour représenter les deux nouveaux types de méthodes.

### Évaluation

On ne peut pas vraiment parler de reconfigurations dans le cas de DART, ou en tout cas pas de reconfigurations structurelles. Les adaptations possibles concernent plutôt le flot d'exécution, et en particulier le mécanisme de liaison dynamique qui est rendu sensible au contexte. Si cette absence de modifications structurelles assure une certaine consistance structurelle, l'utilisation généralisée de la réflexion comportementale ne permet de fournir aucune **garantie** quant à la sémantique des méthodes exécutées. En ce qui concerne les **performances**, l'utilisation de la réflexion est souvent très coûteuse, et encore plus dans le cas de DART, puisque les réifications et les choix de méthodes sont effectuées à chaque envoie de message, même si rien n'a changé dans le contexte depuis la dernière fois. En contrepartie, cette technique assure une bonne **modularité**, puisque le grain des adaptations est très fin. En ce qui concerne le niveau d'**ouverture** du système, le choix de ne supporter l'adaptation que pour un seul mécanisme (l'envoi de message) pourrait sembler négatif mais ce n'est pas le cas en pratique, puisque le comportement d'un objet est intégralement défini par ce mécanisme. De même, le choix d'une approche réflexive garantie la **transparence** du système pour les programmeurs d'applications, tout en leur permettant si besoin est d'intervenir grâce aux méthodes adaptatives.

Les auteurs mentionnent la présence d'informations contextuelles, sur lesquelles les décisions d'adaptation sont basées, mais ne donnent aucun détail les concernant.

Le système offre le support nécessaire pour intégrer des stratégies d'adaptation, mais aucun pour leur création. Ces stratégies seront donc des programmes « normaux », ce qui exclut la plupart des **analyses** spécifiques (stabilité par exemple). En revanche, puisqu'il s'agit de méta-programmes ils peuvent être totalement **génériques** et associés **dynamiquement** aux objets qu'ils contrôlent.

### 3.4.3 LEAD++

#### Description

**Introduction.** LEAD++ [Amano and Watanabe, 1999] est un langage orienté objet basé sur Java auquel il rajoute des fonctionnalités réflexives pour permettre l'écriture de composants adaptatifs.

**Structure des objets.** LEAD++ est une implémentation d'un modèle plus général appelé DAS[23]. Les éléments de base de ce modèle sont :

- les objet environnementaux (*environmental objects*), qui réifient l'état de l'environnement d'exécution ;
- les objets événements (*event objects*), qui servent à notifier d'autres parties du système lorsque certains changements interviennent dans l'environnement ;
- les procédures adaptables (*adaptable procedures*), qui permettent de spécifier plusieurs implémentation d'une méthode ;
- un mécanisme d'adaptation capable de choisir parmi plusieurs implémentations d'une procédure adaptable la plus adaptée, en fonction d'une stratégie d'adaptation.

**Procédures adaptables.** Une procédure adaptable est en fait un ensemble de méthodes, chacune étant associée à une condition environnementale (le corps de `StateCond` dans l'exemple de la figure 3.5).

```
public adaptable void slotBind(StandardEnv env) {
  StateCond {
    doc: "AC-power. So, com2 is not active.";
    type: boolean;
    body: { return env.power() == "AC-power"; }
  }
  RunCode {
    com2 = null;
  }

  StateCond {
    doc: "Battery. So, com2 is active.";
    type: boolean;
    body: { return env.power() == "Battery"; }
  }
  RunCode {
    com2 = new Com2();
  }
}
```

FIG. 3.5 – Exemple de procédure adaptable LEAD++

**Adaptation.** Le mécanisme d'adaptation est réalisé par une double indirection (*double dispatch*). Lorsqu'une procédure adaptable est invoquée au niveau de base, l'appel est redirigé au niveau méta. Un objet *Stratégie* [Gamma et al., 1994] est alors instancié et invoqué. Cet objet stratégie est lui-même une procédure adaptable (au niveau méta), et choisit une de ses implémentations (dans ce cas, un mécanisme d'exécution parmi plusieurs possibles) en fonction de la configuration du système. Ce mécanisme est alors utilisé pour choisir au niveau de base quelle méthode doit être invoquée en fonction des conditions environnementales qui leur sont associées. On peut voir cet algorithme comme une extension de l'algorithme standard de liaison dynamique des langages objets : au lieu de ne considérer que le type du receveur pour

[23] Dynamically Adaptable Software

choisir la méthode à exécuter (ou les types des arguments dans certains langages), LEAD++ considère le type du receveur *et* les conditions d'exécution courantes.

**Évaluation**

LEAD++ fournit tous les éléments nécessaires pour permettre l'écriture d'applications adaptatives mais les stratégies d'adaptation et les conditions déclenchant les adaptations doivent être codées de façon très explicites par le programmeur.

Tout comme dans le cas de DART, l'adaptation ne se fait pas par reconfiguration structurelle, mais en rendant le flot d'exécution (ici l'appel de méthode) sensible au contexte. Si cela **garantit** que l'interface des objets reste stable et offre un **grain très fin**, rien ne peut garantir la cohérence des différentes implémentations alternatives d'une méthode. Pour que les données modifiées par ces méthodes soient toujours cohérentes, toutes les implémentations doivent avoir les mêmes effets de bord, ce qui limite beaucoup les opportunités d'adaptation. On retrouve d'ailleurs cette contrainte dans ACEEL (cf. Section 3.3.4), qui a l'avantage de l'exprimer explicitement. Le choix de l'implémentation est effectué à chaque appel de procédure, même si rien n'a changé dans l'environnement entre deux appels successifs, ce qui peut nuire gravement aux **performances** dans le cas de procédures appelées fréquemment. Enfin, le système est **fermé** et **statique**, et **non transparent**, toutes les implémentations alternatives étant définies « en dur » dans le même fichier source.

Le contexte d'exécution est supposé réifié sous la forme d'objets normaux dont l'état peut être inspecté dans les *state conditions*. Le système n'offre aucun support pour l'écriture, la gestion ou le raisonnement sur ces objets.

Les stratégies d'adaptation sont encodées dans l'ensemble des *state conditions* d'une procédure adaptable. Cette forme simple semble à première vue propre à être **analysée**, mais le fait que le corps des conditions étant constitué de code arbitraire (plutôt que de prédicats comme dans CARISMA, cf. Section 3.2.6) rend ces analyses impossibles. Comme nous l'avons déjà dit pour les reconfigurations, les stratégies sont définies dans le code source des procédures, ce qui les rend **non génériques** et **statiques**.

## 3.5 Conclusion

Le tableau 3.1 synthétise les évaluations des différents travaux étudiés dans ce chapitre en fonction des critères que nous avions préalablement définis. Les cases marquées « n/a » indiquent soit que le critère correspondant n'est pas applicable, soit que nous n'avons pas assez d'informations pour en faire une évaluation ; lorsqu'une (série de) case(s) est vide, cela indique que la fonctionnalité correspondante n'est pas prise en compte par le système évalué.

| | **Reconfigurations** | | | | | **Contexte** | | | | **Stratégies** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Gar. | Mod. | Perf. | Ouv. | Trans. | Préc. | Rich. | Gén. | Perf. | An. | Gén. | Dyn. |
| Open-ORB | − | + | − | + | + | | | | | | | |
| QuO | − | + | n/a | + | ± | n/a | + | ± | n/a | + | − | − |
| dynamicTAO | + | − | n/a | − | ± | n/a | ± | − | n/a | | | |
| MCF[24] | | | | | | n/a | n/a | n/a | n/a | ± | + | − |
| ScalAgent | + | − | n/a | + | ± | + | + | − | n/a | + | ± | − |
| CARISMA | + | − | − | − | + | n/a | − | − | n/a | − | + | − |
| QuA | + | + | n/a | − | ± | | | | | ± | − | − |
| Ad. Comp. | + | − | n/a | − | − | n/a | n/a | n/a | n/a | + | − | − |
| MOLèNE | + | + | n/a | − | − | n/a | + | + | + | + | − | − |
| K-Components | + | + | n/a | + | + | | | | | + | + | ± |
| ACEEL | + | + | n/a | − | − | n/a | − | + | + | + | − | + |
| PLASMA | + | − | + | − | − | *n/a* | + | − | + | + | − | − |
| Odyssey | − | − | + | − | − | n/a | − | − | n/a | + | − | − |
| DART | − | + | − | + | + | n/a | n/a | n/a | n/a | − | + | + |
| LEAD++ | − | + | − | + | + | | | | | − | − | − |

TAB. 3.1 – Résumé des évaluations

Au delà de ces évaluations au cas par cas, on peut distinguer certaines ressemblances entre systèmes qui correspondent à des choix « stratégiques » dans l'approche du problème de l'adaptation.

**Adaptation *au changement vs* adaptation *à l'action*.** (Terminologie empruntée à [Boinot, 2002].) Les systèmes qui choisissent l'adaptation au changement modifient le logiciel à chaque fois qu'ils détectent une évolution significative du contexte d'exécution. C'est le cas de la plupart des solutions étudiées dans ce chapitre, par exemple OpenORB, dynamicTAO, CARISMA, ACEEL et PLASMA. L'autre possibilité, choisie par DART et LEAD++, consiste à n'adapter une partie du logiciel – typiquement une méthode – que lorsqu'elle est invoquée. Les deux approches ont chacune des avantages et inconvénients. L'adaptation au changement peut être contre-productive si elle effectue de nombreux changements dans des parties du système qui ne sont pas utilisés. D'un autre côté, l'adaptation à l'action nécessite d'exécuter la stratégie d'adaptation à chaque utilisation d'une méthode, même si le contexte n'a pas changé depuis la dernière invocation. L'adaptation au changement est préférable dans le cas d'adaptations lourdes mais peu fréquentes, et l'adaptation à l'action est préférable lorsque les décisions d'adaptation ont un coût limité et qu'elles doivent être remises en cause fréquemment. Parmi tous les systèmes étudiés, seuls les « composants adaptatifs » (Sect. 3.3.1, page 35) supportent les deux modes de fonctionnement. Puisque le choix de l'un ou l'autre mode de fonctionnement dépend des conditions particulières, une solution généraliste au problème de l'adaptation doit être capable de gérer ces deux modes d'adaptation complémentaires.

**Adaptation par modification de la structure *vs* adaptation par modification du flot d'exécution.** Dans la plupart des systèmes étudiés, les adaptations sont réalisées en reconfigurant la structure (architecture) du logiciel, ce qui permet indirectement d'adapter le comportement du logiciel, puisque celui-ci est déterminé par sa structure. Ainsi, remplacer l'implémentation d'un service quelconque du *middleware* dynamicTAO revient à effectuer une série de déconnexions et reconnexions des composants appropriés. Une autre possibilité, choisie par LEAD++ et DART consiste à modifier non pas la structure du système mais directement le flot d'exécution, en rendant celui-ci (ou au moins certains de ses mécanismes) sensible au contexte. Il se trouve que parmi les travaux étudiés ici, tous ceux qui ont choisi l'adaptation au changement ont aussi choisi d'effectuer ces adaptations par des (re-)configurations structurelles, et inversement, ceux qui s'adaptent à l'action le font en modifiant le flot d'exécution (même les composants adaptatifs [Boinot, 2002] lorsqu'ils choisissent ce mode de fonctionnement). Bien que ces choix soient les plus naturels, cela laisse les deux autres combinaisons non explorées :

- L'adaptation du flot d'exécution au changement n'a pas vraiment de sens puisque le(s) flot(s) d'exécution de l'application et les évolutions du contexte sont asynchrones.
- Reste l'adaptation structurelle à l'action, qui peut être intéressante dans certains cas puisqu'elle permet de n'effectuer certaines reconfigurations que lorsque les fonctionnalités correspondantes sont effectivement utilisées. En contrepartie, les reconfigurations structurelles étant typiquement plus lourdes que celles du flots d'exécution, cette combinaison risque d'augmenter le temps de latence des fonctionnalités concernées.

**Code d'adaptation intégré *vs* code d'adaptation séparé.** Les travaux étudiés dans ce chapitre illustrent différents niveaux d'intégration entre le code applicatif et le code spécifique à l'adaptation. Dans certains cas, comme [Boinot, 2002], [Amano and Watanabe, 1999] ou même [Capra et al., 2003] (pour ce qui est du code de participation aux enchères), les stratégies d'adaptation sont intégrées au code applicatif, quitte à rajouter de nouveaux mécanismes au langage de programmation. Cependant, dans la plupart des propositions, le code d'adaptation est séparé du code applicatif : OpenORB encapsule les stratégies dans des *contrôleurs* au niveau méta, QuO et les K-Components introduisent des langages spécifiques pour exprimer les stratégies en dehors du code applicatif, et PLASMA introduit de nouveaux types de composants, séparés des composants métier mais intégrés dans l'application, pour implémenter le code d'adaptation. La liaison entre ces deux aspects peut être plus ou moins lâche, et plus ou moins dynamique. Les systèmes basés sur la réflexion comme OpenORB permettent généralement de modifier la liaison (le « tissage ») dynamiquement, alors que QuO et les K-Components utilisent des techniques de génération de code beaucoup plus statiques.

**Forme des stratégies libre *vs* stratégies à base de règles *vs* algorithme fixe paramétré par des données.** Dans certains cas (OpenORB, dynamicTAO) aucun support particulier n'est fourni pour

définir le code des stratégies d'adaptation, qui sont alors programmées de façon totalement libre. Bien que cette approche permette l'utilisation d'algorithmes éventuellement très sophistiqués, elle empêche toute analyse du comportement des stratégies. À l'inverse, des systèmes comme le Middleware Control Framework, l'extension de ScalAgent ou QuA reposent sur des algorithmes fixes, dont le comportement est configuré par les applications (en spécifiant par exemple les besoins en ressources de tel ou tel composant). Cette approche permet des analyses formelles des algorithmes utilisés (voir à cet égard les preuves fournies par le Middleware Control Framework), mais réduisent considérablement le pouvoir d'expression des « stratégies » d'adaptation. Beaucoup des systèmes étudiés utilisent un compromis entre ces deux extrêmes où les stratégies sont spécifiées sous la forme de règles réactives (le *Fuzzy Control Framework* du *Middleware Control Framework*, les K-Components, ACEEL, QuO, les profils applicatifs de CARISMA, PLASMA...), qui associent à un événement se produisant dans le contexte une modification de l'application. Suivant le niveau de formalisme utilisé dans l'expression des événements et des modifications, ce format permet d'effectuer certaines analyses sur le comportement des stratégies tout en laissant une assez grande liberté dans la création des stratégies.

**Utilisation de modèles objets ou composants standards *vs* création de modèles *ad hoc*.** Enfin, on peut distinguer parmi les systèmes étudiés ceux qui utilisent des modèles objets ou des modèles de composants déjà existants (RM-ODP pour OpenORB, CORBA pour QuO et dynamicTAO, Fractal pour PLASMA) de ceux qui utilisent des modèles *ad hoc* conçus spécifiquement pour leurs besoins (tous les autres, à divers degrés). Bien que l'utilisation de modèles *ad hoc* permette de garantir de bonnes propriétés, cela se fait souvent en limitant le pouvoir d'expression des programmeurs du code applicatif, et en les obligeant à utiliser un modèle non standard et parfois peu naturel (ACEEL, *Adaptive Components*, LEAD++).

**Conclusion.** Étant données les diverses possibilités identifiées ci-dessus, nous décidons de baser notre approche sur les choix suivants :

- Puisque les deux modes d'adaptation, au changement et à l'action ont des avantages et des inconvénients complémentaires, nous choisissons de supporter les deux.
- Concernant les mécanismes de reconfiguration, nous choisissons de ne supporter que les modifications de la structure des applications. En effet, ce type de reconfiguration est le seul compatible avec les deux modes d'adaptation, est potentiellement plus efficace, et plus général puisqu'il permet indirectement d'agir aussi sur le comportement.
- L'objectif final de nos travaux étant de faciliter la création d'application adaptative, nous choisissons de séparer le plus possible le code d'adaptation du code applicatif, et de lier les deux de la façon la plus lâche et dynamique possible. Ce choix nous permet de bénéficier des avantages classiques du développement modulaire dans le cas particulier de l'aspect d'adaptation.
- Nous choisissons d'exprimer les stratégies d'adaptation sous la forme de règles réactives, qui nous semble être le compromis idéal entre la généralité, la puissance d'expression et l'analysabilité. De plus, une approche réactive correspond parfaitement à la nature du processus d'adaptation.
- Enfin, notre objectif étant de fournir une solution la plus générale possible, nous utiliserons un modèle de composants existant plutôt que de créer un modèle nouveau et spécifique. De plus, nous choisirons de préférence un modèle conceptuellement simple au détriment des modèles patrimoniaux généralement rendus très complexes par leur besoins d'interopérabilité.

# Deuxième partie

# Contributions

# Chapitre 4

# Architecture de SAFRAN

## Sommaire



## 4.1 Introduction

La contribution principale de cette thèse est le système SAFRAN (*Self-Adaptive Fractal CompoNents*) qui étend le modèle de composants Fractal afin de faciliter le développement d'applications adaptatives. La conception de SAFRAN est basée sur trois principes fondamentaux :

1. L'utilisation d'un *modèle de composants* dynamique et *réflexif*, Fractal [Bruneton et al., 2004], pour la construction des applications. L'utilisation de composants permet de rendre explicite l'architecture d'une application et encourage le découplage de ses différentes fonctionnalités. Si le modèle

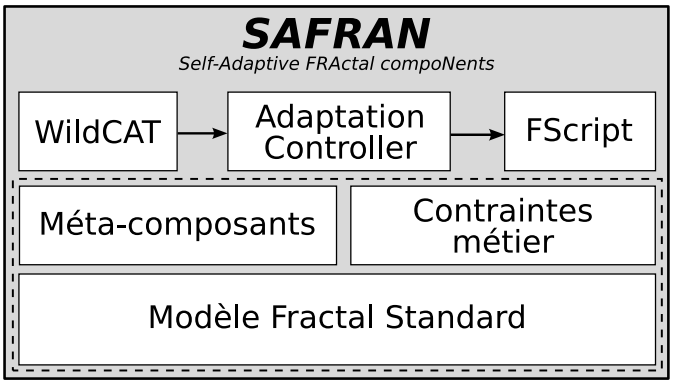


FIG. 4.1 – Les différents éléments constituant le système SAFRAN.

utilisé est dynamique, il est alors possible de reconfigurer l'application en cours d'exécution en modifiant son architecture, et donc indirectement son comportement, afin de l'adapter aux évolutions de son contexte d'exécution. Le caractère réflexif du modèle utilisé est essentiel pour que l'application puisse être reconfigurée – et donc adaptée – de façon non-anticipée[1] [Redmond and Cahill, 2002].

2. Le traitement de la stratégie d'adaptation de l'application comme un *aspect*. Cette approche permet de développer la stratégie d'adaptation séparément du code métier, aussi bien sur le plan spatial que temporel. Le découplage spatial correspond à la modularisation de la stratégie en dehors du code métier, ce qui évite de polluer ce dernier avec des considérations spécifiques à certaines conditions d'exécution, et permet de le rendre plus général et donc plus facilement réutilisable. Le découplage temporel permet de développer la stratégie une fois que l'application est déployée – voire en cours d'exécution – lorsque les conditions d'exécution et les besoins réels sont mieux circonscrits. La stratégie est ensuite composée (tissée) dynamiquement avec les composants métiers et peut être dé-tissée, re-tissée, etc. aussi souvent que nécessaire.
3. Enfin, l'utilisation d'un *langage dédié* pour programmer des *politiques d'adaptation*[2]. Un tel langage permet de définir des politiques de façon beaucoup plus concise et facile à comprendre qu'avec un langage généraliste (même avec l'aide de bibliothèques). De plus, un langage dédié ayant un pouvoir d'expression plus limité qu'un langage généraliste, les politiques pourront éventuellement être étudiées et analysées afin de prédire leurs caractéristiques de comportement (absence d'erreurs, performances, risques de conflits, etc.).

Le modèle de composants Fractal supporte (presque) toutes les fonctionnalités dont nous avons besoin, et est suffisamment extensible pour que nous puissions y intégrer proprement les autres fonctions qui composent SAFRAN. Fractal a donc été choisi comme « substrat » pour la création d'applications *adaptables* (i.e. capables d'être adaptées).

Les principales contributions de cette thèse, sous la forme du système SAFRAN, consistent donc à étendre ce modèle de base afin qu'il supporte les deux autres points ci-dessus : *(i)* le tissage dynamique de politiques d'adaptation dans des composants Fractal, et *(ii)* un langage dédié pour la programmation de ces politiques. Ce dernier est lui-même constitué de plusieurs sous-systèmes qui ont été conçus de façon générique et peuvent donc être considérés comme des contributions à part entière.

La figure 4.1 représente les différents modules qui composent SAFRAN.

– À la base, on retrouve le modèle de composants Fractal, constitué d'un noyau et d'un ensemble d'extensions standards. Nous y avons ajouté deux nouvelles extensions afin d'augmenter les capacités d'adaptation du modèle :

[1] Un modèle de composants comme ArchJava [Aldrich et al., 2002] par exemple supporte des reconfigurations dynamiques mais uniquement si elles sont programmées statiquement.

[2] Une politique d'adaptation peut être vue comme une version concrète, exécutable, d'une *stratégie* d'adaptation abstraite.

  - un protocole à méta-objets pour permettre de réaliser des adaptations non anticipées ;
  - la possibilité de définir des contraintes architecturales spécifiques à une application, au-delà de ce que peut exprimer le système de type de Fractal, comme par exemple le fait qu'un composite donné doit toujours contenir exactement deux sous-composants connectés d'une certaine façon.

  Le modèle de composants enrichi résultant permet de créer des composants *adaptables*, c'est-à-dire souples et reconfigurables tout en conservant leur intégrité – en particulier structurelle. Ce modèle de composants « seulement » adaptables est le substrat sur lequel se base le système SAFRAN pour permettre la construction de composants et d'applications *adaptatives*.
- SAFRAN lui-même est constitué de plusieurs éléments :
  - Le cœur du système est constitué d'une extension générique de Fractal pour le support de politiques d'adaptation dynamique (`AdaptationController`), et d'une implémentation concrète de cette extension. Cette implémentation fournit un langage dédié (appelé lui aussi SAFRAN) pour programmer les politiques sous la forme de règles réactives inspirées du « paradigme » ECA[3] [Dittrich et al., 1995].
  - En amont, ce langagé dédié utilise le système WildCAT pour l'observation du contexte d'exécution de l'application et la notification des événements qui s'y produisent. Concrètement, cela signifie que les événements primitifs mentionnés dans les règles réactives des politiques d'adaptation (la partie $E$ dans ECA) peuvent utiliser toute la puissance de WildCAT pour détecter les modifications significatives du contexte d'exécution.
  - En aval, SAFRAN utilise le langage dédié FScript pour la spécification des reconfigurations qui mettent en œuvre les adaptations. En pratique, les actions des règles réactives ($A$ dans ECA) sont donc des programmes FScript, capables de modifier dynamiquement l'application tout en garantissant son intégrité.

Dans la suite de ce chapitre, nous présentons rapidement le modèle Fractal, puis les deux extensions que nous y avons ajouté. Nous donnons ensuite un aperçu du langage dédié SAFRAN pour la programmation des politiques d'adaptation réactives et l'intégration de ces politiques dans Fractal. Ensuite, nous décrivons les deux sous-systèmes sur lesquels repose l'implémentation des politiques : WildCAT pour l'observation du contexte et FScript pour l'exécution de reconfigurations dynamiques. Nous montrons enfin comment ces différents éléments s'intègrent dans un tout cohérent et comment SAFRAN s'utilise en pratique dans le cycle de développement des applications, avant de conclure en donnant le plan du reste du document où chacune de ces contributions sera détaillée.

## 4.2 Aperçu du modèle de composants Fractal

*[. . . ] program structure should be such as to anticipate its adaptations and modifications. Our program should not only reflect (by structure) our understanding of it, but it should also be clear from its structure what sort of adaptations can be catered for smoothly.*

— Edsger W. Dijkstra

### 4.2.1 Introduction

Cette section présente rapidement Fractal, un modèle de composants développé par France Télécom Recherche & Développement et distribué dans le cadre du consortium ObjectWeb sur lequel nous avons choisi de baser nos travaux. Nous renvoyons le lecteur à la spécification officielle du modèle [Bruneton et al., 2003] pour une description complète.

Les raisons qui nous ont fait choisir Fractal comme modèle de base pour nos travaux sont les suivantes :

1. Fractal est un modèle général, c'est-à-dire qu'il n'est pas spécifique à un domaine d'application. Le modèle des EJB, par exemple, est très orienté vers la construction d'applications de commerce électronique sur le web.

[3] Event–Condition–Action

2. Le modèle de base de Fractal, s'il n'est pas très innovant (il reprend les grandes lignes du modèle RM-ODP), intègre dans un tout *cohérent* les concepts de base communs à la plupart des autres modèles. Cela rend nos travaux relativement généraux et *a priori* applicables – au moins dans le principe – à d'autres modèles, où l'on retrouvera presque toujours les mêmes notions (composants, interfaces, connexions, composition hiérarchique...).
3. Contrairement à d'autres modèles, comme CCM [Wang et al., 2000], COM[4], et les EJBs, dont une grande partie de la complexité vient de besoins d'interopérabilité avec des systèmes ou langages patrimoniaux, Fractal n'est pas encombré d'un lourd bagage historique, ce qui le rend beaucoup plus simple et élégant que d'autres modèles. Bien que l'interopérabilité soit un problème important en soit, il ne fait pas partie en tant que tel de notre problématique. En faisant abstraction de ce problème, Fractal nous permet de nous concentrer sur notre problématique spécifique.
4. Sur le plan technique, le modèle Fractal est très souple, permettant de nombreuses reconfigurations dynamiques (en grande partie grâce à son support de la réflexion), et surtout est conçu à la base pour être facilement étendu et permettre l'ajout de nouvelles fonctionnalités.
5. Enfin, son implémentation de référence, Julia, est très complète, libre, de bonne qualité, et supporte directement toutes les possibilités d'extension permises par le modèle.

Fractal est donc un modèle simple, complet, dynamique et extensible, qui nous offre toutes les fonctionnalités de base dont nous avons besoin sans les rendre plus complexes que nécessaires, et nous permet d'intégrer facilement nos travaux dans le cadre général du modèle lui-même.

### 4.2.2 Anatomie d'un composant Fractal

Une application Fractal est vue comme un assemblage de composants, chacun constitué de deux parties : un *contrôleur* et son *contenu*[5]. Ces composants communiquent deux à deux de façon synchrone en s'envoyant des messages. Les types de message qu'un composant est capable d'envoyer ou de recevoir sont matérialisés par des *interfaces* portées par le contrôleur du composant et visibles de l'extérieur. Le rôle du contrôleur est double :

- interpréter directement certains messages, reçus par ses interfaces dites *de contrôle* et qui permettent d'introspecter et de manipuler le composant en tant que tel ;
- déléguer les autres messages entrant (reçus depuis l'extérieur) à son contenu chargé de les implémenter, et faire parvenir les messages sortant (reçus depuis l'intérieur) vers leur destinataire final. Dans ces deux cas, le contrôleur peut intercepter et manipuler les messages avant de les renvoyer.

La figure 4.2 présente un exemple simple d'application Fractal, constituée de trois composants. Les parties grisées correspondent aux contrôleurs. Les interfaces sont représentées par des formes en « T », les interfaces de contrôle étant sur la partie supérieure du contrôleur, et les interfaces de services sur les bords gauche ou droit, suivant qu'elles sont fournies ou requises par le composant.

Le contenu d'un composant peut être constitué soit d'autres composants, auquel cas on parle de *composant composite*, soit d'un objet (au sens large) du langage de programmation sous-jacent, auquel cas on parle de *composant primitif*. L'exemple de la figure est constitué d'un composite qui contient lui-même deux sous-composants, ces derniers étant primitifs, puiqu'ils ne contiennent pas d'autres composants. Toutes les interactions (envois de messages) entre composants Fractal devant passer par l'intermédiaire des contrôleurs de leurs composants, ceux-ci constituent le lieu privilégié où implémenter les services non-fonctionnels (contrôle de la sécurité, invocation à distance...) dont ont besoin la plupart des applications sophistiquées[6].

[4]Component Object Model

[5]Dans le Kell Calculus [Stefani, 2003], calcul formel dont est inspiré Fractal, le contrôleur est appelé *membrane*, et le contenu *plasme*, par analogie avec les cellules des organismes vivants.

[6]Notons qu'aucune des implémentations de Fractal n'offre actuellement de garanties sur l'intégrité des communications : aucun mécanisme n'empêche les programmeurs ces composants de « court-circuiter » les connexions et encapsulations et d'utiliser les mécanismes du langage sous-jacent sans contraintes. Cette limitation est d'ailleurs une des pistes de recherche futures pour améliorer le modèle et ses implémentations.

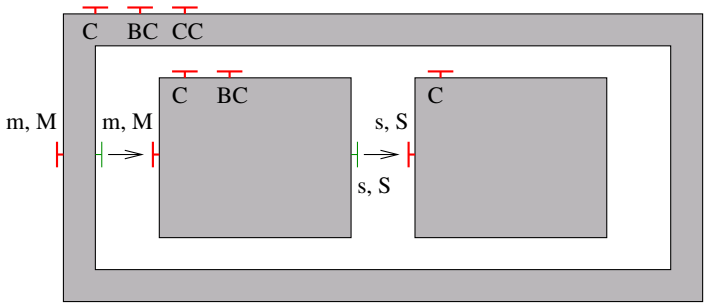


FIG. 4.2 – Anatomie d'un composant Fractal.

Conformément à ce modèle général, Fractal fournit deux mécanismes de structuration des applications permettant de rendre explicite leur architecture : les connexions entre interfaces de composants (représentées par des flèches sur la figure) matérialisent un canal de communication entre ces deux interfaces, et une relation de contenance permet de regrouper un ensemble fini de composants dans un composite, qui peut ensuite être manipulé comme une « boîte noire ». La relation de contenance définie par Fractal a ceci de particulier – par rapport à la plupart des autres modèles – qu'elle n'est pas stricte, c'est-à-dire qu'un composant donné peut être contenu dans plusieurs composites (tant que cela ne crée pas de cycles). Graphiquement, cette relation se traduit tout simplement en représentant les sous-composants à l'intérieur du contenu du composite. En réalité, le modèle Fractal se voulant très général, ces deux mécanismes ne sont pas « codés en dur » dans la spécification. Il est ainsi possible de ne pas les utiliser, ou bien d'étendre le modèle en ajoutant d'autres types de relations entre composants, qui seront alors traitées de la même manière que les relations prédéfinies. Voir par exemple notre extension pour la réflexion comportementale décrite dans la section 4.3.1.

### 4.2.3 Le noyau du modèle et les interfaces de contrôle standards

Le modèle Fractal est construit autour d'un noyau minimal de concepts fondamentaux, que l'on peut voir comme l'intersection des concepts présents dans la quasi-totalité des modèles de composants. Puisque l'un des objectifs principaux de Fractal est de permettre la gestion et la reconfiguration dynamique des architectures logicielles, ces différents concepts doivent être explicites pendant l'exécution. Ils sont donc décrits sous la forme d'interfaces de programmation (API[7]), en utilisant une syntaxe proche de celle des interfaces Java avec quelques restrictions, et en remplaçant les types primitifs spécifiques à Java par des équivalents plus génériques (`any` et `string` remplaçant respectivement `Object` et `String`). L'objectif est d'obtenir une spécification du modèle indépendante d'un langage particulier, tout en permettant de dériver facilement et systématiquement une API spécifique à un langage donné[8]. La figure 4.3 représente le noyau du modèle Fractal sous la forme d'un diagramme de classes UML[9].

On peut résumer ce noyau de la façon suivante : un *composant* possède un ensemble d'*interfaces* identifiées par un nom, les composants comme les interfaces étant *typés*. Ce noyau est donc très général et pourrait être utilisé comme cadre pour construire des modèles de composants très divers. Le modèle standard décrit dans la spécification de Fractal ajoute à ce noyau un ensemble d'interfaces de contrôle standards qui permettent d'introspecter et de manipuler les différents fonctionnalités des composants :

- L'interface `AttributeController` permet d'exposer de façon standardisée les paramètres de configuration dynamique d'un composant en utilisant un mécanisme inspiré des JavaBeans.
- Les interfaces `ContentController` et `SuperController` permettent de manipuler le contenu des composants composites et de naviguer dans la hiérarchie de composition correspondante.

[7] Application Programming Interface
[8] Actuellement, les seuls « *mappings* » concrets spécifiés sont ceux des langages Java et C.
[9] Unified Modeling Language

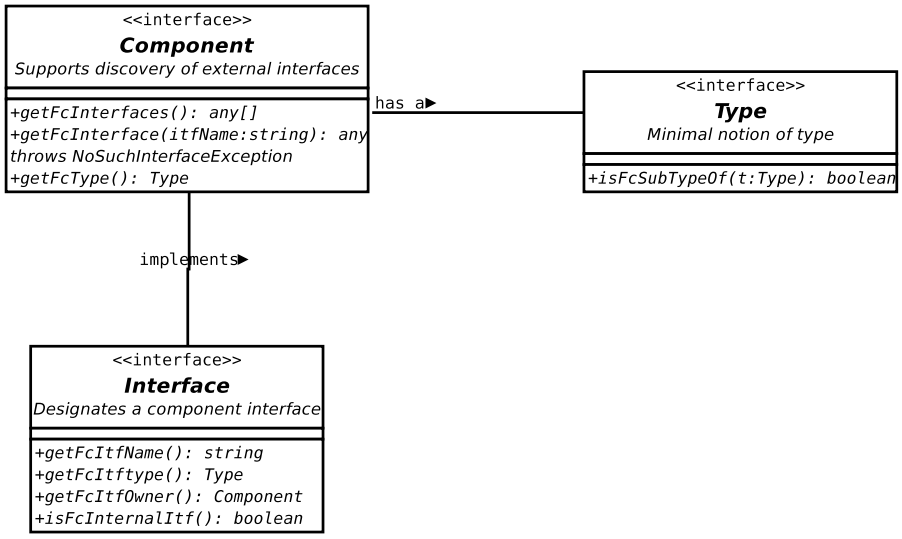


FIG. 4.3 – Le noyau du modèle de composants Fractal.

- L'interface `BindingController` permet d'introspecter et de manipuler les connexions entre interfaces de composants.
- Enfin, l'interface `LifeCycleController` permet de gérer le cycle de vie (minimaliste) des composants, qui peuvent être soit *démarrés* soit *stoppés*.

La spécification officielle de Fractal définit aussi un système de type standard pour les interfaces et les composants, ainsi qu'un protocole d'instanciation des composants basé sur le motif de conception *Fabrique* [Gamma et al., 1994]. Nous renvoyons le lecteur à la spécification pour ces détails, qui ne sont pas essentiels à la compréhension du reste de ce document.

## 4.3 Extensions de Fractal pour l'adaptabilité

Comme illustré par la figure 4.1, SAFRAN repose sur une version étendue du modèle de composants Fractal standard. Afin d'augmenter l'adaptabilité des composants Fractal, nous avons en effet créé deux nouvelles extensions :

- La première est un protocole à méta-objet (MOP) adapté au modèle Fractal, qui permet d'adapter des composants Fractal de façon non-anticipée [Redmond and Cahill, 2002].
- La seconde ajoute la possibilité d'associer à un composant Fractal des *contraintes architecturales*, qui expriment des critères de consistance spécifiques à la sémantique d'un composant particulier, au-delà de ce que permet le système de type de Fractal. Ainsi, il devient possible d'exprimer le fait qu'un composite ne doit jamais être vide, ou bien qu'il doit contenir exactement deux composants et que ces derniers doivent être connectés d'une certaine façon.

On peut considérer que le MOP augmente la flexibilité des composants Fractal, et donc leur possibilité d'adaptation, alors que le support des contraintes métier renforce la structure de l'application en restreignant l'ensemble de reconfigurations acceptables. Il s'agit donc encore ici de trouver le bon équilibre entre forme et ouverture en ce qui concerne la puissance des adaptations possibles.

### 4.3.1 Méta-composants Fractal

Le modèle de composant Fractal tel que décrit jusqu'à maintenant offre une grande partie des fonctionnalités nécessaires à la création d'applications adaptatives. En particulier, il supporte la réflexion structurelle qui permet la reconfiguration dynamique de l'architecture de l'application. Cependant, Frac-

tal ne supporte pas la réflexion comportementale [Maes, 1987], qui permet de modifier le comportement d'un composant de façon générique. Or, cette fonctionnalité est nécessaire pour permettre la réalisation d'adaptations comportementales non-anticipées [Redmond and Cahill, 2002]. Nous avons donc dû étendre le modèle standard de Fractal pour y ajouter cette fonctionnalité, sous la forme d'un protocole à méta-objets (MOP) [Kiczales et al., 1991] très simple, mais adapté au modèle de composants. Pour réaliser cette extension, nous avons utilisé les possibilités de configuration et d'extension de Julia, l'implémentation de référence du modèle Fractal en Java.

Par rapport à la plupart des langages à objets, l'introduction d'un MOP dans Fractal est considérablement simplifiée par la présence dans chaque composant d'un contrôleur. En effet, la principale difficulté technique rencontrée lors de l'extension d'un langage pour la réflexion est l'introduction d'un niveau d'indirection – en l'occurence de réification – qui n'a pas été prévu. Dans le cas de Fractal, toutes les interactions entre composants se font par envoi de message aux interfaces de composants, et tous ces messages, entrant ou sortant sont sous le contrôle du contrôleur de composant. Puisque dans Julia, l'implémentation de référence, ces contrôleurs sont justement prévus pour être très extensibles, l'ajout d'un lien méta [Maes, 1987] dans Fractal entre parfaitement dans le cadre du modèle général.

La conception de notre protocole est très simple. Par rapport au modèle standard décrit ci-dessus, nous n'ajoutons que deux notions : la notion de méta-composant, similaire à celle de méta-objet ; et la possibilité d'associer par un *lien méta* un composant quelconque, dit « de base » à un méta-composant. La notion de lien méta ne fait pas partie du noyau du modèle, mais elle a exactement le même statut que la relation de composition entre un composite et ses sous-composants, ou les connexions entre interfaces.

Pour qu'un composant Fractal puisse être associé à un méta-composant, il doit avoir une interface de contrôle particulière, dénommée `metalink-controller` :

```
package org.obasco.fractal.mop;
interface MetaLinkController {
  void setFcMetaComponent(Component meta) throws NoSuchInterfaceException;
  Component getFcMetaComponent();
}
```

Cette interface permet de connaître le méta-composant associé au composant de base, s'il y en a un, et de modifier cette association. La méthode `setFcMetaComponent()` lève une exception si le composant passé en paramètre n'est pas un méta-composant valide. Un tel méta-composant se distingue d'un composant normal par la présence d'une interface `message-handler`, qui lui permet de contrôler le comportement des composants de base auxquels il est associé. La signature de cette interface est la suivante :

```
package org.obasco.fractal.mop;
interface MessageHandler {
  any invoke(string itfName, any target, any meth, any[] args) throws InvocationTargetException;
}
```

Puisque toutes les interactions entre composants se font par envoi de message, seul ce mécanisme doit être contrôlé par un méta-composant. Lorsqu'un méta-composant est associé à un composant de base, tous les messages reçus par ce dernier sont réifiés par son contrôleur et redirigés vers la méthode `invoke()` de l'interface `message-handler` du méta-composant. Le méta-composant est libre d'interpréter le message réifié de n'importe quelle façon, y compris en le renvoyant vers sa cible originale. Les paramètres de cette méthode sont :

- `itfName` : le nom de l'interface à laquelle le message de base est destinée.
- `any` : la cible à laquelle le message est destinée. Si le composant de base est un composant primitif, cette cible est l'objet Java qu'il contient. Si le composant est un composite, cette cible est l'interface du sous-composant connectée à l'interface externe qui a reçu le message.
- `meth` identifie la méthode spécifique invoquée. Dans le cas de Java, ce paramètre est un objet de classe `java.lang.reflect.Method`.
- `args` est la liste des paramètres du message.

Notre MOP ne prend pas directement en compte la possibilité d'associer plusieurs méta-composants à un même composant de base. Cependant, il est parfaitement possible de créer des méta-composants composites, contenant d'autres méta-composants. La composition des comportements de ces sous-composants est cependant à la charge du composite. De même, un méta-composant donné n'est pas nécessairement lié à un seul composant de base ; le protocole de l'interface `message-handler` réifie suffisamment d'informations pour qu'un méta-composant puisse distinguer au cas par cas les messages destinés à différents composants de base. Le lien méta est donc de cardinalité $n$-$m$. De plus, notre conception permet aussi très naturellement d'avoir plusieurs niveaux méta : rien n'empêche un méta-composant d'avoir une interface `metalink-controller` et d'être lui-même contrôlé par des méta-méta-composants.

Pour conclure, cette extension permet d'adapter de façon transparente un composant Fractal de base en modifiant la façon dont sont interprétés les messages destinés à ses interfaces de services. Il devient alors possible par exemple d'ajouter dynamiquement un méta-composant qui met en cache les résultats de certaines opérations jugées trop coûteuses [David and Ledoux, 2003], ou bien encore de dupliquer les messages reçus par un composant pour en maintenir un réplicat actif.

### 4.3.2 Spécification de contraintes architecturales métiers

Le modèle Fractal définit un ensemble de contraintes architecturales générales qui déterminent si une configuration est valide ou non, en particulier à travers son système de type. Par exemple, pour qu'un composant soit utilisable – état `STARTED` – toutes ses interfaces requises obligatoires doivent être connectées à des interfaces compatibles.

Cependant, pour une application donnée, il est possible d'effectuer des reconfigurations valides du point de vue de Fractal mais qui laissent l'application dans un état qui ne lui permet pas de fonctionner normalement. Les contraintes définies au niveau du modèle ne sont pas assez expressives pour refléter les spécificités d'une application particulière.

Nous avons défini une nouvelle extension de Fractal qui permet d'associer à un composant un ensemble de *contraintes* spécifiques à la sémantique de l'application. Ces contraintes sont définies par des expressions dans le langage dédié FPath, puis associées dynamiquement aux composants.

FPath, est un langage inspiré d'XPath [World Wide Web Consortium, 1999], la différence principale étant que là où XPath permet de naviguer dans des documents XML, FPath permet lui de naviguer dans des architectures Fractal et d'y sélectionner des composants. De la même manière que XPath modélise un document XML sous la forme de nœuds reliés par des axes, FPath repose sur une modélisation des composants Fractal sous la forme d'un graphe similaire. Un exemple d'expression FPath est `child::server/attribute::*` qui renvoie tous les paramètres de configuration du sous-composant dénommé `server`. Un autre exemple pourrait être `count(child::*[interface::task]) < 10`, qui vérifie qu'un composite ne possède pas plus de dix sous-composants ayant une interface `task`. FPath sera décrit en détail dans la section 6.3.

Les contraintes architecturales utilisées par notre extension sont donc des expressions FPath relatives au composant auquel elles sont associées et évaluées pour leur valeur booléenne. Lorsque la valeur d'une des expressions est `false`, cela signifie que la contrainte métier correspondante est violée, et que la configuration en cours, bien que consistante du point de vue de Fractal, est incorrecte du point de vue de la sémantique de l'application.

Notre extension se présente sous la forme d'une nouvelle interface de contrôle optionnelle qui permet d'ajouter et de retirer dynamiquement des contraintes à un composant, et de tester la valeur d'une contrainte donnée. La signature de cette interface est la suivante :

```
interface ConstraintsController {
  void     addFcConstraint(string name, string expr) throws IllegalArgumentException;
  void     removeFcConstraint(string name);
  string   getFcConstraint(string name);
  string[] getFcConstraints();
  boolean  evaluateFcConstraint(string name) throws IllegalArgumentException;
  string[] getFcViolatedConstraints();
```

```
}
```

Le fonctionnement de ces différentes méthodes est le suivant :

- `addFcConstraint(name, expr)` permet d'ajouter une nouvelle contrainte à un composant. Toutes les contraintes d'un composant sont identifiées par un nom logique, indiqué ici par le paramètre `name`, qui doit être unique pour un composant donné. La contrainte elle-même est indiquée par le paramètre `expr` sous la forme d'une chaîne qui doit représenter une expression FPath. Une erreur de type `IllegalArgumentException` est levée s'il existe déjà une contrainte portant le nom `name` associée au composant, ou bien si l'expression est incorrecte (erreur de syntaxe, ou utilisation de fonctions inconnues par exemple).
- `removeFcConstraint(name)` permet de retirer une contrainte étant donné son nom `name`. Si aucune contrainte de ce nom n'existe, la méthode n'a aucun effet.
- `getFcConstraint(name)` permet de retrouver l'expression FPath de la contrainte à partir de son nom `name`. Si aucune contrainte n'existe sous ce nom, la méthode renvoie `null`.
- `getFcConstraints()` renvoie un tableau contenant les *noms* de toutes les contraintes actuellement associées au composant.
- `evaluateFcConstraint(name)` évalue la contrainte indiquée dans le contexte de l'état courant du composant, et renvoie sa valeur après conversion en booléen. Si la méthode renvoie `true`, cela signifie que la contrainte est actuellement vérifiée ; sinon, cela signifie qu'elle est violée (mais n'indique pas pourquoi). Une erreur de type `IllegalArgumentException` est levée si aucune contrainte nommée `name` n'est actuellement associée au composant.
- `getFcViolatedConstraint()` évalue tour-à-tour chacune des contraintes actuellement associées au composant, et renvoie un tableau contenant les noms des toutes celles qui sont fausses/violées. Renvoie `null` (et non pas un tableau vide) si toutes les contraintes sont vérifiées.

Notons que dans la version actuelle de notre implémentation, les contraintes associées à un composant n'ont qu'une valeur informative. En effet, puisque n'importe quelle reconfiguration impliquant le composant peut *a priori* modifier la valeur d'une contrainte, le système devrait re-vérifier *toutes les contraintes* de *tous les composants impliqués* à chaque reconfiguration, ce qui est prohibitif en terme de performances. De plus, même si une violation est détectée, les APIs de Fractal ne sont pas conçues pour notifier ce genre d'erreur, et les prendre en compte conduirait à modifier Fractal de façon incompatible (par exemple en ajoutant un nouveau type d'erreur, `ConstraintException`, à quasiment toutes les méthodes). La seule autre possibilité serait d'analyser chaque expression FPath utilisée pour déduire l'ensemble des méthodes qui peuvent en changer la valeur, mais cette solution est trop complexe à mettre en œuvre et de plus, rien ne garantit qu'elle permettrait de gagner beaucoup en performances.

En pratique, c'est aux clients intéressés de vérifier la valeur des contraintes si et quand ils le veulent. C'est le cas de FScript, qui prend en compte les contraintes associées aux composants impliqués dans les reconfigurations pour décider *a posteriori* si une reconfiguration est valide ou non (en complément de critères de consistance plus généraux).

## 4.4 Politiques d'adaptation SAFRAN

Nous présentons dans un premier temps l'extension de Fractal qui permet d'associer dynamiquement une ou plusieurs politiques d'adaptation à un composant Fractal, puis nous décrivons la structure des politiques d'adaptation elles-mêmes. Enfin, nous justifions en quoi SAFRAN peut être considéré comme un système à aspects, et ce qui le différencie des autres systèmes.

### 4.4.1 Un contrôleur Fractal pour gérer l'adaptation

SAFRAN introduit une extension au modèle Fractal qui permet d'associer (tisser) dynamiquement des politiques d'adaptation aux composants métiers d'une application. Cette extension est conçue pour être minimale et générique, et son interface ne dépend donc pas de la forme exacte des politiques utilisées. Ainsi, bien que SAFRAN fournisse un langage dédié pour la programmation de politiques d'adaptation,

d'autres approches sont possibles et peuvent utiliser l'interface décrite ici (par exemple des politiques programmées de façon *ad hoc*, ou bien des agents logiciels).

Comme toutes les extensions du modèle Fractal, celle-ci se traduit par la définition d'une nouvelle interface de contrôle, qui sera implémentée par les composants SAFRAN. C'est la présence de cette interface qui en fait des *composants adaptatifs* : là où un composant Fractal standard fournit des interfaces de contrôle pour permettre à un acteur externe de le reconfigurer (et donc de l'adapter), un composant SAFRAN intègre grâce à cette interface le code d'adaptation et devient ainsi adaptatif, i.e. acteur de sa propre adaptation.

Si elle est présente, cette interface de contrôle doit être identifiée sous le nom `adaptation-controller` et avoir la signature suivante :

```
interface AdaptationController {
  void attachFcPolicy(any policy) throws InvalidPolicyException;
  void detachFcPolicf(any policy) throws NoSuchPolicyyException;
  any[] getFcPolicies();
}
```

Les méthodes de cette interface permettent d'inspecter et de manipuler dynamiquement l'ensemble des politiques d'adaptation qui s'appliquent au composant :

- `attachFcPolicy()` permet d'attacher une nouvelle politique au composant. Puisque notre extension est à ce stade générique et doit pouvoir être utilisée avec différents types de politiques, le type du paramètre est `any`. En pratique, une implémentation donnée de cette interface, comme par exemple SAFRAN, n'accepte qu'un type bien défini de politiques. Cette méthode lève une exception de type `InvalidPolicyException` si la politique ne peut pas être ajoutée au composant. Cela peut se produire si l'implémentation de la politique ne correspond pas à celle de l'interface, ou bien par exemple si la nouvelle politique est incompatible avec celles déjà associées au composant. S'il n'y a pas d'erreur, la nouvelle politique prend effet immédiatement et jusqu'à ce qu'elle soit retirée.
- `detachFcPolicy()` permet de détacher une politique du composant afin qu'elle ne s'y applique plus. Une exception de type `NoSuchPolicyException` est levée si la politique passée en paramètre n'est pas associée au composant.
- Enfin, `getFcPolicies()` permet de connaître l'ensemble des politiques actuellement associées au composant. Cette spécification abstraite de l'interface n'indique pas l'ordre dans lequel les politiques sont renvoyées, mais une implémentation particulière peut spécifier cet ordre s'il a un sens dans son cas.

Lorsqu'une politique est associée à un composant donné, elle est chargée d'adapter ce composant, et uniquement celui-ci. En effet, l'un des intérêts principaux de la conception par composants est d'isoler très fortement les différents éléments d'une application, en ne les laissant interagir que par l'intermédiaire d'interfaces bien définies. L'ajout dans une application de politiques d'adaptation ne remet pas en cause cette conception, et une politique associée à un composant donné ne doit pas être capable de modifier les autres parties de l'application. Si cela était le cas, il deviendrait impossible de raisonner localement sur les effets d'une politique sans connaître l'intégralité de l'application. Nous introduisons donc une règle qui restreint la portée des politiques, quelle que soit son implémentation : une politique d'adaptation liée à un composant $c$ n'a le droit de *modifier* que ce composant-ci et ses sous-composants directs s'il s'agit d'un composite. Ainsi, l'ajout d'une politique à un composant ne brise pas l'encapsulation. Cette règle peut sembler au premier abord très restrictive, surtout en regard de ce que nous avons dit à propos de l'aspect d'adaptation, qui est souvent transverse à la décomposition de l'application. Cependant, le support du partage de composants par le modèle Fractal associé à la possibilité de créer des composites à l'exécution permet facilement de créer des politiques d'adaptation qui agissent sur des composants autrement dispersés dans le code, tout en respectant la restriction ci-dessus. Il suffit pour cela de regrouper tous les composants à adapter dans un composite auquel sera associée la politique[10]. Ce nouveau composite,

[10]On retrouve un mécanisme similaire dans les *politiques applicatives* du système décrit dans [David and Ledoux, 2002], qui permet de regrouper des objets selon certains critères afin de les adapter de la même façon.

qui ne sert qu'à regrouper des composants, matérialise le fait que ces composants ont quelque chose en commun (un aspect) qui impose de les adapter de façon synchronisée.

### 4.4.2 Structure de politiques d'adaptation

L'extension de Fractal présentée ci-dessus décrit de façon abstraite la séparation et la composition de l'aspect d'adaptation sans expliquer comment ce dernier est spécifié. L'implémentation concrète des politiques d'adaptation que fournit SAFRAN se présente sous la forme d'un langage dédié.

La nature de ces politiques SAFRAN (forme et sémantique) découle de notre analyse du domaine de l'adaptation et plus particulièrement du *processus d'adaptation*. En effet, jusqu'à présent, nous avons identifié, analysé (Sect. 1.2) et évalué (Chapitre 3) les différents éléments qui constituent un logiciel adaptatif séparément : *(i)* sensibilité au contexte, *(ii)* présence de mécanismes de reconfiguration dynamique, et *(iii)* politiques d'adaptation. Cependant, cette vision statique n'est pas suffisante, car l'adaptation est avant tout un *processus*, qui modifie continuellement l'application pendant son exécution pour qu'elle soit à tout moment la mieux *adaptée* aux circonstances présentes.

D'une façon générale, on peut décomposer ce processus en trois étapes successives :

1. *Observation* de l'environnement (contexte d'exécution) de l'application et/ou de l'application elle-même, et *détection* de l'occurence de certaines circonstances nécessitant une adaptation. Cette étape peut être implémentée grâce à la *sensibilité au contexte* d'une application adaptative.
2. *Prise de décision* : étant données d'une part l'état actuel de l'application et d'autre part les nouvelles circonstances détectées dans la première phase, l'application doit décider des opérations de reconfiguration à effectuer pour s'adapter aux nouvelles circonstances. Cette étape correspond à la stratégie d'adaptation de l'application.
3. *Action* : une fois que les opérations de reconfiguration ont été déterminées, il reste à les appliquer concrètement à l'application, tout en évitant de perturber son bon fonctionnement « normal ». Cette étape correspond aux *mécanismes de reconfiguration dynamique* de l'application adaptative.

Une politique d'adaptation doit donc être *réactive*, détecter les *événements* significatifs qui se produisent dans le contexte, et en fonction de ces événements et de l'état actuel du système, *agir* sur l'application en la reconfigurant pour l'adapter. Afin de répondre à tous ces besoins, les politiques d'adaptation de SAFRAN sont structurées sous la forme d'ensembles de *règles réactives* de la forme

```
when <event>
if   <condition>
do   <action>
```

Ce type de règles est similaire à ce que l'on peut trouver dans le domaine des bases de données actives [Dittrich et al., 1995; Collet, 1996] sous la dénomination de règles ECA : *Event, Condition, Action*. Une règle d'adaptation indique que *lorsque* un événement correspondant à l'expression `<event>` se produit, *si* l'expression `<condition>` concernant l'état du système est vraie, *alors* la reconfiguration `<action>` est exécutée, modifiant ainsi le système afin de l'adapter à la nouvelle situation résultant de l'occurence de l'événement.

La spécification de l'événement correspond à la fonctionnalité d'observation et de détection des changements du contexte d'exécution. La condition et l'action spécifient comment réaliser l'adaptation. La règle dans son ensemble, qui relie un événement à une action (conditionnelle) correspond à la prise de décision.

Dans le système SAFRAN, les politiques d'adaptation associées dynamiquement aux composants Fractal adaptatifs sont constituées de séquences (ordonnées) de règles d'adaptation ;

```
policy example() = {
  rule { when <event1> if <cond1> do <action1> };
  rule { when <event2> if <cond2> do <action2> };
  rule { when <event3> if <cond3> do <action3> };
```

```
  ...
}
```

Lorsqu'une telle politique est associée à un composant SAFRAN grâce à son interface `adaptation--controller`, le contrôleur du composant se met à l'écoute des événements mentionnés dans la politique. Dès qu'un de ces événements se produit, la ou les règles correspondantes sont activées, leurs conditions évaluées, et pour celle qui sont vraies, leurs actions exécutées en séquence.

### 4.4.3 SAFRAN en tant que système à aspects

Le point de départ de notre approche est de considérer l'adaptation comme une *aspect*, c'est-à-dire une préoccupation transverse mais néanmoins modularisée grâce à des techniques de composition (tissage) appropriées (cf. Section 2.1). Un aspect est constitué d'un ensemble de couples $(coupe, action)$, où une coupe est un ensemble de *points de jonction*. [Douence et al., 2002] ont montré que ces points de jonction pouvaient être vus comme de *événements* relatifs à l'exécution du programme. Jusqu'à aujourd'hui, presque tous les systèmes à aspects ont choisi de ne considérer que des événements au niveau du langage de programmation : invocation de fonction ou de méthode, création d'objet, etc.[11] Or, une application n'est jamais totalement isolée : elle est affectée par le contexte dans lequel elle s'exécute et avec lequel elle interagit (cf. Section 1.1).

SAFRAN se distingue des autres systèmes à aspects par l'extension du domaine des points de jonction à l'ensemble du contexte de l'application. Les événements *endogènes* (liés à l'exécution du programme lui-même) détectables par SAFRAN correspondent ainsi aux points de jonction « traditionnels », alors que les événements *exogènes* (liés au contexte d'exécution) constituent un nouveau type de points de jonction qui permet de réagir aux évolutions du contexte, enrichissant par là même le pouvoir d'expression des aspects.

En ce qui concerne les *actions* (*advices*) déclenchées par la détection d'un point de jonction, la plupart des systèmes à aspects sont génériques et les aspects qu'ils permettent de programmer ne sont pas liés à un domaine spécifique. En conséquence, leurs actions sont en général programmées avec le même langage que le programme de base (Java pour AspectJ par exemple), parfois légèrement étendu (fonction spéciale `proceed()` en AspectJ). Dans notre cas particulier, le domaine est restreint aux adaptations par reconfiguration de l'architecture. Ainsi, le *langage* SAFRAN peut être considéré aussi bien comme un DSL que comme un ASL[12], le domaine ou la classe d'aspects considéré étant l'adaptation. SAFRAN fournit des constructions spécifiques pour exprimer les coupes (événements endogènes et exogènes) et un langage dédié aux actions (FScript).

Une fois qu'un aspect a été programmé, ici sous la forme d'une politique d'adaptation, il doit être *intégré*, ou tissé dans l'application à laquelle il est destiné. Dans le cas de SAFRAN, les politiques sont tissées et détissées dynamiquement dans le contrôleur d'adaptation de leur composant cible (cf. figure 4.4). En effet, le contrôleur d'un composant Fractal est destiné à gérer toutes les interactions entre ce composant et son environnement. Dans le cadre du modèle Fractal standard, cela signifie essentiellement gérer les messages entrant et sortant, mais puisque SAFRAN étend justement cette notion d'environnement afin d'y inclure le contexte d'exécution, il est normal que le code destiné à gérer ces interactions, c'est-à-dire les politiques d'adaptation, soient « tissées » dans le contrôleur du composant. Notons tout de même que le verbe « tisser » n'implique pas, comme par exemple c'est le cas pour AspectJ, de transformation de code ; le contrôleur d'un composant Fractal, en particulier lorsqu'il est implémenté avec Julia, offre déjà les niveaux d'indirection nécessaires pour y intégrer du code facilement.

Une fois qu'une politique (aspect) est intégrée (tissée) dans le contrôleur d'un composant, ce dernier devient *adaptatif*. L'exécution du code correspondant est réactive : lorsque des événements, qu'ils soient endogènes ou exogènes, sont reçus par le contrôleur d'adaptation, ce dernier détermine la réaction appropriée en fonction des différentes régles présentes à cet instant, puis exécute cette réaction afin d'adapter

[11] Il existe aussi quelques travaux comme [Ostermann et al., 2005] qui essayent d'exprimer des coupes sémantiques, plus abstraites et moins dépendantes des mécanismes du langage sous-jacent.

[12] Aspect-Specific Language

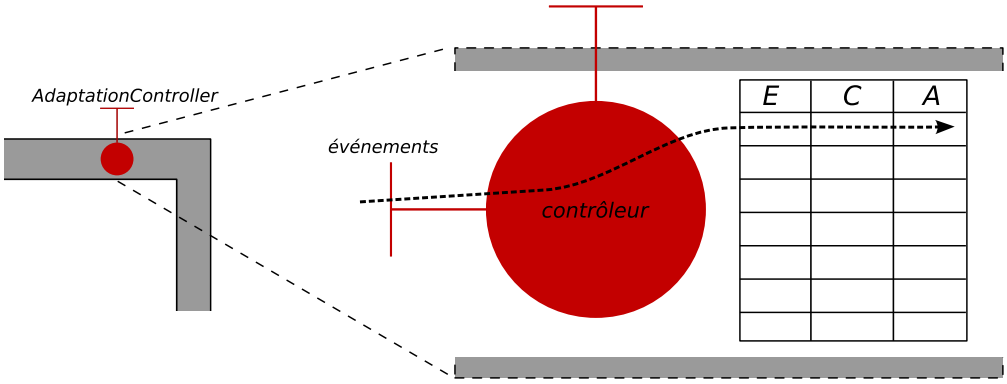


FIG. 4.4 – Intégration des politiques dans le contrôleur d'un composant adaptatif.

le composant aux nouvelles circonstances (cf. Section 7.4 pour les détails du modèle d'exécution). Ce schéma d'exécution est en accord avec la nature réactive du processus d'adaptation, dont il reprend les différentes phases : observation, décision, puis action.

## 4.5 Sensibilité au contexte avec WildCAT

WildCAT est un système léger mais puissant pour la création d'applications sensibles au contexte (*context-aware*) [Dey and Abowd, 2000]. Ce système est utilisé par SAFRAN pour implémenter la partie $E$ (événement) des règles d'adaptation qui nécessite d'observer le contexte d'exécution et de détecter les événements mentionnés, qui doivent déclencher des adaptations. Bien que WildCAT ait été conçu dans le cadre de SAFRAN, il s'agit d'une contribution à part, indépendante. WildCAT se présente sous la forme d'un framework général et extensible distribué sous une licence libre, et donc réutilisable dans d'autres domaines et d'autres projets. L'objectif est de permettre le partage du code d'observation du contexte et de détection des changements. En effet, ce code souvent de bas niveau (interaction avec le système d'exploitation, voire le matériel) nécessite parfois une grande expertise pour être implémenter efficacement. Puisqu'une grande partie de ce code est commun à de nombreuses applications, il est très intéressant de pouvoir le partager et le réutiliser. En offrant une interface claire et générale, WildCAT offre un cadre dans lequel ce partage peut se réaliser. Du point de vue du programmeur d'application, WildCAT se présente comme une bibliothèque Java traditionnelle, qui permet grâce à son interface d'accéder au contexte d'exécution de l'application. Étant donné l'objectif de généralité de WildCAT, ce contexte est représenté par un modèle de données simple et léger mais très général. L'interface de programmation de WildCAT permet de découvrir les caractéristiques du contexte, de raisonner sur celui-ci et d'être notifié automatiquement lorsque des événements particuliers se produisent.

### 4.5.1 Modélisation du contexte d'exécution

Le contexte d'exécution est organisé en un ensemble de *domaines contextuels*, identifiés par un nom unique, qui représentent chacun un aspect spécifique du contexte, par exemple ressources matérielles, ressources logicielles, réseau (topologie, performances...), contexte géo-physique (position géographique, température ambiante...), profil utilisateur (ses préférences, ses (in-)capacités, son activité en cours...), etc. Chacun de ces domaines contextuels est modélisé sous la forme d'une arborescence de *ressources* nommées, chacune d'entre elle étant décrite par un ensemble d'*attributs* (de simples paires $(nom, valeur)$).

WildCAT fournit une syntaxe simple, inspirée de celle des URI[13], pour désigner les ressources et attributs : `domaine://chemin/vers/une/ressource#attribut` (la partie `#attribut` étant optionnelle). Par exemple :

[13] Uniform Resource Identifier

– `sys://storage/drives/hdc#removable` désigne un attribut booléen qui indique si le disque correspondant est amovible (vrai pour les lecteurs de CD-ROM et DVD-ROM, faux pour les disques durs) ;
– `geo://location/physical#latitude` et `geo://location/physical#longitude` indiquent la position actuelle de l'ordinateur hôte ;
– La ressource `geo://location/logical` et ses attributs `#building`, `#floor` et `#room` indiquent aussi la position de l'hôte, mais d'une façon différente.

De la même manière que le contexte d'une application change pendant son exécution, le modèle de ce contexte fournit par WildCAT évolue dynamiquement : les valeurs des attributs peuvent changer, attributs et ressources peuvent apparaître et disparaître à tout moment. Ainsi, lorsque l'utilisateur branche une souris USB sur son ordinateur portable, une nouvelle ressource `sys://input/mouse` apparaît dans le modèle WildCAT avec tous les attributs appropriés pour la décrire (`#vendor`, `#dpi`...), voire même éventuellement des sous-ressources comme `position` ou `buttons`.

### 4.5.2 Interface de programmation

Du point de vue du programmeur qui utilise WildCAT, le contexte est représenté par un unique objet Java, instance de la classe `Context`. Cet objet peut être utilisé pour accéder au contexte suivant deux modes de fonctionnement complémentaires :

1. Le premier (mode *pull*) permet au programme de découvrir la structure du contexte et d'exécuter des requêtes sur son état courant. Le programmeur peut ainsi obtenir la liste des domaines contextuels, ou bien étant donné un chemin désignant une ressource, obtenir la liste de toutes ses sous-ressources et de ses attributs, y compris les valeurs de ces derniers. Cette interface est utilisée afin de permettre aux conditions et aux actions (parties $C$ et $A$ des règles SAFRAN) de dépendre de l'état du contexte au moment de leur activation.
2. La seconde interface (mode *push*) permet au programme de s'abonner à différents types d'événements. Une fois abonné, le programme est ensuite notifié de façon asynchrone à chaque fois que l'un des événements en question se produit. C'est cette interface qui est utilisée pour implémenter la partie $E$ des règles d'adaptation SAFRAN : une fois déployée sur un composant, une politique d'adaptation s'abonne auprès de WildCAT afin d'être notifiée de l'occurrence des événements mentionnés dans les règles d'adaptation.

### 4.5.3 Événements SAFRAN

Toutes les modifications du modèle WildCAT, qui reflètent des modifications dans le contexte d'exécution de l'application elle-même, se traduisent par la génération d'*événements primitifs* qu'il est possible de référencer dans les règles d'adaptation SAFRAN. Ainsi, si une politique d'adaptation nécessite de reconfigurer l'application lorsque la mémoire disponible devient trop faible, la règle d'adaptation correspondante pourra spécifier comme événement l'expression :

```
sys://storage/memory#free < 10_000_000
```

SAFRAN délègue l'observation de cet attribut et la détection du dépassement de seuil à WildCAT, qui notifie le(s) composant(s) concerné(s) dès que la condition se réalise. Lorsque la règle en question est déployée dans le cadre d'une politique d'adaptation SAFRAN, le contrôleur d'adaptation du composant correspondant s'abonne auprès de WildCAT pour être notifié de l'événement en question. Lorsque l'événement se produit finalement, WildCAT prévient le contrôleur en lui indiquant toutes les informations nécessaires pour décrire cette occurrence particulière de l'événement (dans l'exemple précédent, la valeur exacte de `sys://storage/memory#free` qui a déclenché l'événement). Le contrôleur dispose alors de toutes les informations nécessaires pour exécuter la ou les règles d'adaptation déclenchées. Cette séparation des tâches permet aux contrôleurs d'adaptation de se concentrer sur l'exécution des politiques d'adaptation sans avoir à se préoccuper des détails techniques de l'observation du contexte.

## 4.6 Reconfigurations dynamiques consistantes avec FScript

Bien qu'il supporte toutes les fonctionnalités dont nous avons besoin, le modèle Fractal est spécifié sous la forme d'un ensemble d'interfaces de programmation (APIs). Cette forme ne convient pas à notre mode d'utilisation : il est impossible d'offrir la moindre garantie quant à la consistance des reconfigurations effectuées, le code correspondant devient rapidement très verbeux et incompréhensible, et les concepts spécifiques à Fractal n'existent pas directement en Java. Ces limitations nous ont conduit à la création de FScript, un langage dédié pour la spécification et l'exécution de reconfigurations structurelles d'applications et de composants Fractal.

En dehors des aspects purement syntaxiques (notations spécifiques à Fractal), FScript se distingue d'un langage de script « normal » par les *garanties* qu'il offre concernant la consistance des reconfigurations. En effet, les reconfigurations FScript étant destinées à adapter des applications en cours d'exécution, nous devons garantir que les reconfigurations ne risquent pas de rendre l'application inutilisable. Pour cela, nous avons choisi un certain nombre de critères de consistance, en particulier l'*intégrité transactionnelle* (atomicité, consistance de l'état final, isolation) et la *terminaison en temps borné* des reconfigurations. La validation de ces critères est garantie d'une part par la structure même du langage, dont le pouvoir d'expression est volontairement limité, et d'autre part par son implémentation, qui effectue des vérifications dynamiques.

### 4.6.1 Expressions FPath

FPath est un sous-langage de FScript qui correspond aux expressions qui permettent d'observer l'architecture de l'application, mais pas de la modifier. La distinction entre d'une part les expressions FPath purement fonctionnelles et d'autre part le reste du langage FScript qui peut modifier l'application cible permet de réutiliser FPath dans d'autres contextes (par exemple les contraintes métier ou bien les conditions des règles réactives) dans lesquels cette absence d'effet de bord est importante.

FPath se présente sous la forme d'une notation simple et expressive, inspiré d'XPath [World Wide Web Consortium, 1999], pour la navigation dans une architecture Fractal et la sélection d'éléments (composants, interfaces, attributs) répondant à certains critères. Le langage repose sur la modélisation d'un ensemble de composants Fractal sous la forme d'un graphe orienté, dont les nœuds représentent les composants, leurs interfaces et attributs, et dont les arcs sont annotés par des *labels* qui dénotent le type de relation entre deux nœuds (interface, sous-composant, connexion...). En plus des types d'expressions habituelles (arithmétique, combinateurs booléens et comparaisons, etc.), FPath ajoute des expressions de type *chemins relatifs* (à un composant de départ). Un chemin consiste en une suite de pas, chacun constitué de trois éléments : `axe::test[predicat]`. À chaque pas, un ensemble de nœuds de départ est converti en un nouvel ensemble en suivant les arcs du graphe identifié par l'axe, puis en filtrant le résultat grâce au *test* et aux *prédicats* optionnels.

Par exemple l'expression FPath `child::client/binding::server` sélectionne dans un premier temps le sous-composant (axe `child`) nommé `client` (test) du composant initial, puis suit la connexion (axe `binding`) dont le nom est `server` pour finalement retourner l'interface serveur correspondante. De même, l'expression `count(interface::*[required(.) and not(bound(.))]) > 0` renvoie *vrai* si et seulement si le composant initial possède des interfaces requises qui ne sont pas encore connectées.

### 4.6.2 Programmes FScript

FScript ajoute à FPath la possibilité de définir des *actions de reconfiguration*, en combinant expressions FPath, structures de contrôles simples et manipulation de variables. Toutes les opérations de reconfiguration dynamique supportées par Fractal (voir la section 4.2) sont disponibles pour les programmes FPath sous la formes d'actions prédéfinies, dites primitives. L'exemple suivant montre la définition de deux actions de reconfiguration FScript qui pourraient être utilisées pour adapter des composants SAFRAN.

```
action increase-volume(player, amount) = {
  vol := $player/@volume + $amount;
```

```
  if ($vol < 100) then {
    set-value($player/@volume, $vol);
  } else {
    set-value($player/@volume, 100);
  }
}

action select-strategy(cache, strat) = {
  if ($cache/binding::strategy != $strat) then {
    itf := $cache/interface::strategy;
    if (bound($itf)) then {
      previous := $cache/binding::strategy;
      unbind($itf);
      stop($previous);
    }
    bind($itf, $strat/interface::replacement-strategy);
    start($strat);
  }
}
```

La première action augmente le volume sonore d'un lecteur multimédia, jusqu'à une certaine limite en manipulant le paramètre de configuration correspondant, et pourrait être utilisée par exemple pour réagir à une augmentation du bruit ambiant. La seconde action permet de modifier la stratégie de remplacement utilisée par un composant cache (par exemple pour passe de LRU[14] à LFU[15]) en modifiant la connexion qui relie le composant cache au composant stratégie.

Si jamais une erreur se produit pendant l'exécution d'une de ces actions, par exemple si la nouvelle stratégie de remplacement indiquée (paramètre `strat`) n'est pas du bon type, faisant échouer la connexion `bind(...)`, l'implémentation de FScript s'assure que le système retourne dans son état initial avant l'exécution de l'action (*atomicité*). De même, si l'action `increase-volume` modifie le volume sonore en dehors des limites imposées par une contrainte métier (par exemple `@volume > 25`), FScript considèrera que l'action a échouée, même si elle est valide du point de vue du modèle Fractal. L'action sera aussi annulée dans ce cas (*consistance* de l'état final).

## 4.7 Modèle de développement d'applications adaptatives

Nous avons à présent décrit tous les éléments qui constituent le système SAFRAN ; il nous reste à indiquer comment ce dernier s'utilise en pratique et surtout comment il s'intègre dans le processus de développement d'applications « classique » afin de réaliser l'objectif final de notre travail : rendre plus simple le développement d'applications adaptatives elles-mêmes plus efficaces.

### 4.7.1 Modèle de développement standard

Afin de tirer partie au mieux des possibilités de réutilisation offertes par les composants logiciels, les applications à base de composants sont généralement conçues selon une approche « *top-down* », du plus général au plus précis : *(i)* dans un premier temps, seule l'*architecture* globale de l'application est spécifiée ; *(ii)* celle-ci est ensuite *implémentée*, soit en réutilisant des composants « sur l'étagère » (COTS), soit en créant de nouveaux composants spécifiques, soit en considérant un composant comme une sous-application, et en raffinant son architecture récursivement, jusqu'à ce que l'application soit complètement implémentée et empaquetée ; *(iii)* l'application implémentée peut ensuite être *déployée*, c'est-à-dire *installée* et *configurée*, *exécutée*, et enfin *administrée* (mises à jour, évolutions...).

[14] Least Recently Used
[15] Least Frequently Used

### 4.7.2 Étapes supplémentaires introduites par SAFRAN

Le modèle de développement décrit ci-dessus considère qu'une application est uniquement constituée d'un assemblage de composants métiers. Puisque l'approche choisie par SAFRAN est justement de séparer le plus possible la logique d'adaptation du code métier, SAFRAN introduit de nouveaux artefacts, de nouveaux rôles et de nouvelles étapes dans le modèle de développement. Le modèle de développement des politiques d'adaptation SAFRAN elles-mêmes est le suivant : *(i)* tout d'abord, les opportunités d'adaptation doivent être *identifiées*, sous la forme d'un ensemble de scénarios qui constituent la stratégie d'adaptation globale de l'application ; *(ii)* ces différents scénarios sont alors *implémentés* par des politiques d'adaptation SAFRAN, et éventuellement des sondes WildCAT ; *(iii)* ces politiques et les sondes associées sont alors *déployées*, les politiques étant attachées aux composants Fractal appropriés, puis éventuellement détachées.

### 4.7.3 Intégration de SAFRAN dans le modèle standard

Nous avons décrit ci-dessus les étapes de développement des composants métier d'une part et des politiques d'adaptation d'autre part, sachant que ces deux aspects – le « programme de base » et l'aspect d'adaptation – sont destinés à être intégrés (tissés) pour finalement constituer l'*application adaptative*. SAFRAN permet d'effectuer cette intégration à différentes étapes du développement, selon le type de scénario d'adaptation. À chaque étape du modèle de développement décrit ci-dessus, le contexte d'utilisation final de l'application devient de mieux en mieux connu (i.e. la variabilité spatiale se réduit de plus en plus), même s'il reste souvent très dynamique (variabilités temporelle et des besoins). De composants génériques et réutilisables, on passe à des composants spécialisés pour une application, puis à une application destinée à un site particulier, et enfin à des tâches spécifiques. La nature des politiques d'adaptation qui peuvent être intégrées à chaque étape dépend beaucoup de cette connaissance du contexte d'exécution.

1. Bien que ce ne soit pas le cas d'utilisation le plus courant, SAFRAN permet la création de politiques d'adaptation dès le développement de l'application. Ce type de politiques peut être utilisé par exemple pour encapsuler le code de configuration (*customisation*) d'un composant sur l'étagère réutilisé dans l'application, ou bien encore pour décrire des contraintes dynamiques entre plusieurs composants (par exemple « le paramètre `foo` du composant $A$ doit être le double du paramètre `ba` de $B$ »). De par leur nature dynamique et réactive, ces politiques SAFRAN ne seront effectivement attachées qu'au démarrage de l'application. L'approche par aspect choisie par SAFRAN a ici surtout l'intéret de permettre une bonne séparation « physique » des préoccupations en isolant le code générique et réutilisable des adaptations spécifiques à l'application, en l'occurence ici des configurations.
2. Au moment du déploiement sur un site donné, le contexte d'exécution (caractéristiques matérielles principales du ou des hôtes, besoins spécifiques de l'utilisateur concerné, systèmes périphériques existants) sont beaucoup mieux connus, même s'ils peuvent rester très dynamiques. D'autres politiques d'adaptation peuvent alors être attachées aux composants, qui seront ainsi adaptés dynamiquement aux évolutions du contexte qui peuvent être anticipées à ce moment du déploiement.
3. Enfin, pendant l'exécution de l'application, la nature dynamique et réflexive de Fractal et de SAFRAN permet d'ajouter, de supprimer ou de remplacer les politiques d'adaptation à chaque fois que les besoins de l'utilisateur évoluent ou que le contexte change d'une façon qui n'était pas prévu par les politiques initiales.

## 4.8 Plan du reste du document

Dans la suite de ce document, nous reprenons tour à tour chacune des contributions qui composent le système SAFRAN en les décrivant plus en détail. Le chapitre 5 décrit ainsi comment le système WildCAT permet de construire facilement des applications sensibles à leur contexte, et le chapitre 6 présente en détails le langage FScript qui permet de spécifier et d'exécuter des reconfigurations dynamiques sûres de

composants Fractal. Nous décrivons ensuite dans le chapitre 7 le langage dédié qui permet à SAFRAN de programmer des politiques d'adaptation réactives et de les associer dynamiquement à des composants Fractal. Ce langage est construit en partie au dessus des deux contributions précédentes, bien que ces dernières aient été conçues pour être plus générales et applicables en dehors du cadre de SAFRAN. Enfin, le chapitre 8 montre comment l'utilisation du système SAFRAN dans sa globalité s'intègre dans le cycle de développement des applications en l'appliquant à plusieurs cas d'utilisation concrets.

# Chapitre 5

# WildCAT : un outil pour le développement d'applications sensibles au contexte

**Sommaire**



Ce chapitre décrit WildCAT, un système permettant à une application Java d'observer son contexte d'exécution et d'être prévenue automatiquement lorsque certaines circonstances – définies par l'application – se produisent. Cette capacité à percevoir son environnement afin de pouvoir réagir à ses évolutions est indispensable pour une application adaptative, puisque c'est en réponse à ces changements que l'application décide de s'adapter.

## 5.1 Introduction

Fondamentalement, toute adaptation d'un système est une modification de ce système en réponse à un changement de son contexte d'utilisation, avec pour objectif que le système résultant soit mieux adapté

au nouveau contexte (cf. Sect. 1.2, page 10). Avant même de se poser les questions des reconfigurations à appliquer au système, et de la façon de les décrire (qui seront les sujets des chapitres suivants), nous devons résoudre le problème de la perception du contexte d'exécution par l'application.

Le terme « contexte » est très vague, et défini de manières différentes et incompatibles selon les auteurs, quand il est défini. Certains se contentent d'en donner un synonyme, comme « environnement ». D'autres [Brown, 1998] ont une vision restreinte qui n'inclut par exemple que le contexte physique (localisation géographique, heure, température...). Dans [Dey and Abowd, 2000] les auteurs présentent et critiquent diverses définitions existantes dans la littérature, mais celle à laquelle ils aboutissent (« *Context is any information that can be used to characterize the situation of an entity.* ») est beaucoup trop vague et difficilement utilisable en pratique pour décider si une information fait ou non partie du contexte. Dans [Henricksen et al., 2002], les auteurs tentent de dégager une définition à partir d'une étude de cas. Cependant, ils ne font qu'énumérer certaines caractéristiques des informations contextuelles (dynamicité, imperfection des mesures, corrélations entre informations...) et, encore une fois, ne donnent aucun critère opérationnel pour déterminer quelles sont les informations qui font partie du contexte.

Dans ce document, nous adopterons la définition suivante : *le contexte d'exécution d'une application regroupe tous les éléments externes à l'application – autres que ses entrées explicites – qui influencent la qualité de son fonctionnement telle que perçue par son utilisateur.* Cette influence peut se faire sentir par exemple au niveau des performances, de la sécurité, ou des fonctionnalités. Les « éléments » qui font partie du contexte d'une application peuvent être eux aussi de natures très diverses : les autres logiciels interagissant avec l'application, la machine hôte qui l'exécute, l'environnement physique de celui-ci, le ou les utilisateur(s), etc.

Cette définition du contexte met en évidence le fait que celui-ci est déterminé par l'application. Ainsi, si le niveau de bruit ambiant dans une pièce ne fait pas partie du contexte d'un tableur, il fait partie de celui d'un lecteur multimédia, qui peut par exemple ajuster son volume sonore pour rester audible à son utilisateur.

Notons que notre définition, contrairement à celles des articles cités ci-dessus, fournit un critère opérationnel simple pour déterminer si une entité fait ou non partie du contexte d'une application.

La qualité de fonctionnement perçue par l'utilisateur inclut les critères de performances temporels (ex : temps de réponse maximal) et spatiaux (ex : utilisation de la mémoire) généralement désignés par l'expression *Quality of Service* (*QoS*), mais n'est pas limitée à ceux-ci. Par exemple, la taille des fontes ou les couleurs utilisées par une application graphique peut être très importante pour son utilisabilité, surtout si son utilisateur a des problèmes de vision (myopie, daltonisme...). Ce genre de critères n'est en général pas pris en compte par les systèmes gérant la Qualité de Service, entre autre car ils ne se prêtent pas facilement à des mesures objectives. Notre définition du contexte d'exécution fait référence à la qualité *perçue par l'utilisateur* du système (toujours un humain), la seule réellement importante au bout du compte. Il est donc nécessaire d'inclure dans le contexte d'exécution des éléments subjectifs comme les préférences de l'utilisateur. Bien entendu, inclure ce genre d'information dans le contexte rend plus difficile l'évaluation de la qualité d'une adaptation, mais il s'agit d'un tout autre problème. Dans ce chapitre, nous nous préoccupons uniquement de permettre aux applications de *percevoir* leur contexte ; ce qu'elles font de ces informations relève d'une autre problématique.

Puisque le contexte d'exécution est externe à l'application et qu'il ne fait pas partie de ses entrées, il n'est généralement pas explicitement représenté dans l'application. Or, puisqu'une application adaptative doit être consciente de son contexte (*context-aware*) pour pouvoir s'y adapter, nous devons trouver un moyen pour observer le contexte et de rendre le résultat cette observation accessible à l'application afin qu'elle puisse raisonner dessus.

Généralement, lorsqu'une application a besoin de s'adapter à un élément de son contexte, elle implémente cette observation de façon spécifique à son cas particulier. Par exemple, un programme qui veut utiliser une taille de fonte appropriée à la résolution de l'écran de l'utilisateur devra interroger le système d'exploitation ou de fenêtrage pour obtenir cette information. Cette approche pose plusieurs problèmes :

- le code de détection correspondant doit être réécrit par chaque application ;
- ce code nécessite souvent d'interagir directement avec le système d'exploitation, voire avec le matériel, ce qui réduit la portabilité de l'application ;

– si l'élément du contexte dont l'application dépend peut évoluer pendant l'exécution de l'application, le code nécessaire pour réagir dynamiquement rend l'application beaucoup plus complexe, et pour cette raison est souvent omis (obligeant par exemple l'utilisateur à redémarrer l'application pour prendre en compte les nouveaux paramètres) ;
– les programmeurs d'une application ont rarement les compétences techniques pour écrire correctement et efficacement ce genre de code de bas niveau, qui nécessite une grande expertise système ; ce ne devrait d'ailleurs pas être leur travail.

La solution que nous proposons, et qui est décrite dans ce chapitre, consiste à fournir aux programmeurs d'application un service générique et réutilisable d'observation du contexte d'exécution. Le rôle de ce service est d'offrir aux applications qui l'utilisent une vision *précise*, *complète* et *correcte* de leur contexte d'exécution, leur permettant de raisonner sur celui-ci et de détecter l'occurrence de certaines circonstances, spécifiques à leur application. Cette approche résout les problèmes cités plus haut, et offre en outre d'autres avantages :

– étant donné qu'il s'agit d'un système à part entière, développé une fois pour toute, il peut être beaucoup plus sophistiqué que les solutions *ad hoc* généralement simplistes développées pour résoudre un problème ponctuel ;
– la modularisation et l'encapsulation de ce service permettent son enrichissement indépendamment des applications qui l'utilisent, tant que l'interface entre les deux reste la même ;
– ce type de service peut être utilisé pour d'autres besoins que l'adaptation d'application, comme par exemple la surveillance ou la collecte de statistiques.

Dans ce chapitre, nous décrivons la conception, l'utilisation et la mise en œuvre de WildCAT, un service générique d'observation du contexte d'exécution. Nous indiquons pour commencer les objectifs et les critères d'évaluation d'un service d'observation du contexte (Section 5.2) avant de présenter le système que nous proposons, WildCAT. Nous décrivons tout d'abord le modèle de données utilisé pour représenter le contexte (Section 5.3), puis l'interface de programmation (API) permettant aux applications d'utiliser ce service (Section 5.4), et le format des fichiers de configurations qui permettent d'instancier le modèle générique pour les besoins spécifiques d'une application (Section 5.5). Nous donnons ensuite quelques détails les possibilités d'extension du *framework* (Section 5.6), avant de conclure le chapitre (Section 5.7).

## 5.2 Objectifs et critères d'évaluation du service

Vu de l'extérieur, un service d'observation du contexte doit offrir une vision structurée des connaissances qu'il a du contexte. Ces informations doivent être disponibles pour l'application à travers une interface bien définie, qui doit répondre à trois besoins principaux :

**Découverte.** L'application doit pouvoir interroger le service pour *découvrir* la structure du contexte, c'est-à-dire quels sont les éléments qui le constitue. Par exemple, « L'utilisateur dispose-t-il d'une souris ? ».

**Interrogation.** L'application doit pouvoir interroger ponctuellement le service pour obtenir *immédiatement* la valeur courante d'une propriété du contexte, connaissant déjà l'existence de cette propriété. Par exemple, si une souris est présente, « Combien a-t-elle de boutons ? ».

**Abonnement et notification.** [1] L'application doit pouvoir s'enregistrer auprès du service pour être notifiée de façon asynchrone lorsque certaines circonstances se produisent. Par exemple, « Préviens moi si l'utilisateur branche ou débranche un périphérique d'entrée (souris, clavier, manette de jeux...) ».

Les différents critères d'évaluation inspirés de l'état de l'art du domaine [Courtrai et al., 2003; Henricksen et al., 2002; Dey and Abowd, 2000] qui nous ont servi pour guider la conception de WildCAT sont les suivants :

**Configurabilité.** Puisque, par définition, ce qui fait partie du contexte dépend de l'application, le service d'observation du contexte doit pouvoir être configuré pour chaque application. Par exemple, les

[1] Parfois aussi appelé Souscription / Publication (en anglais *Publish / Subscribe*).

informations concernant le trafic réseau font partie du contexte d'un serveur web, mais pas de celui d'un traitement de texte. À l'inverse, la taille et la résolution de l'écran sont importants pour ajuster l'interface du traitement de texte, mais n'ont aucun impact sur le serveur web. Chaque application doit pouvoir configurer le service pour observer tous les éléments de son contexte, et uniquement ceux-ci.

**Généralité.** Le modèle de données utilisé pour représenter le contexte de façon structurée doit être le plus générique possible. Notre objectif n'est pas de fournir un service limité à un nombre prédéfini de domaines, comme par exemple les préférences utilisateur, ou bien les caractéristiques des périphériques physiques présents sur une machine. De tels systèmes existent déjà, par exemple GConf[2] et HAL[3] pour les deux exemples de domaines cités. Les concepts proposés par le modèle doivent pouvoir être utilisés pour représenter tous les différents *domaines contextuels* potentiels.

**Richesse du modèle.** Au delà de la généricité, le modèle de données doit être suffisamment riche pour pouvoir représenter tous les aspects pertinents du contexte. Ce besoin de richesse concerne aussi bien les possibilités de structuration des données entre elles, que les types de données possibles et leurs caractéristiques.

**Pouvoir d'abstraction.** Le modèle doit permettre d'offrir une vision plus ou moins abstraite et simplifiée du contexte pour les clients du service. Ainsi, si certaines applications peuvent être intéressées par des détails très précis concernant le trafic réseau (nombre de paquets reçus et perdus par exemple), la plupart peuvent se contenter d'un indicateur moins précis mais plus simple à appréhender (un niveau de qualité sur une échelle de 1 à 10 par exemple).

**Dynamicité.** Si certains éléments du contexte changent rarement (par exemple la localisation géographique d'un serveur), d'autres évoluent continuellement (quantité de mémoire disponible, périphériques présents...). Le service d'observation du contexte doit prendre en compte cet aspect dynamique, qui peut se manifester aussi bien au niveau des valeurs d'une propriété du contexte qu'au niveau structurel (apparition ou disparition de certains éléments).

**Pouvoir d'expression à la disposition des clients.** L'interface de programmation que fournit ce service à ses clients doit être suffisamment expressive pour découvrir les éléments présents, interroger le service concernant une propriété donnée, et s'enregistrer auprès du service pour être notifié de l'occurrence de certains événements (voir la liste des besoins définis plus haut).

**Disponibilité d'une implémentation.** Plus pragmatiquement, le service d'observation du contexte doit exister concrètement sous la forme d'une implémentation utilisable, et pas seulement d'une spécification plus ou moins précise.

**Non-interférence de l'implémentation.** L'exécution du service doit avoir un impact minimal sur le fonctionnement du système observé [Schroeder, 1995]. En particulier, il doit être le plus efficace possible afin d'utiliser le moins de ressources possibles (processeur, mémoire, etc.), d'une part pour permettre aux applications de disposer du maximum de ces ressources, et d'autre part pour ne pas interférer avec ses propres mesures (« effet Heisenberg »).

**Facilité d'utilisation.** En plus de l'interface qu'il offre aux programmes clients, le service, puisqu'il est configurable, doit offrir une interface de configuration. Celle-ci doit permettre aux programmeurs d'une application de spécifier exactement quels éléments font partie du contexte de cette application, sans qu'ils aient à devenir des experts dans des domaines qui ne sont pas les leurs. Cette interface doit donc à la fois permettre d'accéder à toute la richesse du service, tout en restant facile d'accès.

En plus de ces critères qui concernent les aspects génériques du service et de son implémentation, d'autres critères déterminent la qualité des configurations (ou instanciations) particulières effectivement utilisées par les applications :

**Correction des mesures.** Les informations fournies par le service doivent bien entendu être correctes. Si une mesure n'est pas certaine, il vaut mieux donner une valeur approchée, en indiquant qu'il s'agit d'une approximation, qu'une valeur précise mais fausse.

[2] `http://www.gnome.org/projects/gconf/`
[3] `http://www.freedesktop.org/Software/hal`

**Précision des mesures.** Les observations du contexte effectuées par le service doivent être le plus précises possibles. Notons que correction et précision ne désignent pas la même chose. Dans le cas d'une mesure numérique par exemple, une valeur de $50 \pm 10\%$ peut être tout aussi correcte qu'une mesure de $52,5$, mais est moins précise. Un autre aspect de la précision concerne la fréquence des mesures concernant le paramètres évoluant au cours du temps. Plus ces mesures seront fréquentes, plus les informations fournies aux clients seront précises concernant l'évolution du contexte.

**Richesse de la modélisation.** La modélisation doit prendre en compte le plus possible des éléments du contexte de l'application, et des aspects pertinents de ces éléments.

Parmi ces critères détaillés, on retrouve bien tous les critères généraux concernant la conscience de son contexte d'exécution qu'un logiciel adaptatif doit avoir, tels que nous les avions identifiés dans a section 1.2.3, page 12 : **précision**, **richesse**, **généralité et modularité**, et enfin **performances**.

Notons que certains de ces critères sont incompatibles ou du moins contradictoires :

**Configurabilité et pouvoir d'expression *vs.* facilité d'utilisation.** Plus le modèle de données disponible pour décrire le contexte est riche, plus il est complexe à appréhender pour les programmeurs qui veulent utiliser le service dans leur application. Il faut conserver à l'esprit que les programmeurs d'applications sensibles au contexte ne doivent pas être obligés de devenir des experts dans ce domaine.

**Généricité *vs.* richesse du modèle.** Dans une certaine mesure, l'enrichissement du modèle de données peut le rendre moins général, en introduisant des notions qui n'ont de sens que pour certains domaines. D'un autre côté, un modèle trop générique risque de n'avoir aucune sémantique propre, et donc de laisser l'interprétation des données entièrement à la charge des clients.

**Non-interférence de l'implémentation *vs.* tous les autres.** Quasiment tous les critères (richesse du modèle, dynamicité, correction et précision des mesures) nécessitent pour être mis en œuvre une implémentation sophistiquée qui utilise beaucoup de ressources (temps de calcul, espace mémoire ou disque). Si ces ressources sont utilisées par le service, elles ne sont donc plus utilisables par l'application, ce qui nuit à son bon fonctionnement.

Le résultat final ne pourra donc être qu'un compromis, que l'on espère cependant adapté au plus grand nombre de cas d'utilisations possibles. Comme nous le verrons plus loin, WildCAT est conçu spécifiquement pour permettre à chaque application l'utilisant de choisir le compromis qui lui convient le mieux.

## 5.3 Le modèle de données WildCAT

Cette section décrit le modèle de données utilisé par WildCAT pour représenter le contexte d'une application. Ce modèle est conçu en fonction des critères définis dans la section 5.2, en particulier la **généralité**, la **dynamicité** et la **facilité d'utilisation**. L'objectif de ce modèle est de pouvoir représenter l'ensemble des éléments faisant partie du contexte de l'application, avec leurs caractéristiques propres, ainsi que leurs relations les uns avec les autres.

### 5.3.1 Structuration des données

#### Domaines contextuels

WildCAT modélise le contexte d'une application sous la forme d'un ensemble de *domaines contextuels*, chacun représentant un aspect particulier du contexte. Chaque domaine contextuel est identifié par un nom globalement unique dans le contexte d'une application donnée. On peut citer comme exemples de domaines contextuels possibles :

- **sys** : les ressources matérielles disponibles pour l'application, et leur niveau d'utilisation ;
- **soft** : les ressources logicielles disponibles (applications et bibliothèques disponibles, avec leur numéro de version par exemple) ;
- **pref** : les préférences utilisateur et autres paramètres de configuration ;

– **net** : l'état du ou des réseau(x) dont fait partie le système (topologie, performances, niveau d'utilisation...) ;
– **phys** : l'environnement physique de l'hôte sur lequel tourne l'application (localisation géographique, température, bruit et luminosité ambiante...) ;
– **user** : les caractéristiques physiques de l'utilisateur (handicaps en particulier), son activité, son état émotionnel...

La distinction entre les différents domaines contextuels permet de bien séparer les différentes préoccupations, et de **réutiliser** la définition de certains domaines entre plusieurs applications, comme par exemple sys, réduisant d'autant le coût d'utilisation du service pour les programmeurs. Chacun peut être vu comme un *ontologie* modélisant un aspect particulier du contexte de l'application, et peut être développé séparément des autres par des experts du domaine en question. De plus, comme nous le verrons plus loin, cette séparation entre les domaines permet d'offrir des implémentations spécifiques et **efficaces** à chacun d'entre eux, ce qui est primordial vis-à-vis du critère de non-interférence, qui requiert une implémentation efficace du service.

### Structuration des domaines contextuels : notion de ressource

Chaque domaine contextuel est organisé en une arborescence de *ressources* enracinée dans le domaine contextuel, chacune étant identifiée par un nom, unique parmi les sous-ressources directes d'une ressource donnée.

Cette notion de ressource se veut très générale, et peut être utilisée soit pour représenter des éléments « concrets » faisant partie du contexte, comme par exemple un écran ou un disque dur, tout comme des éléments plus abstraits comme la localisation géographique de l'utilisateur final, voire tout simplement pour regrouper plusieurs autres ressources dans une catégorie logique, comme par exemple une ressource storage regroupant les différents disques disponibles. Suivant les cas, la relation entre une ressource donnée et ses sous-ressources dans l'arborescence peut donc représenter une notion de contenance physique (une ressource building contenant des ressources room), de contenance logique (des préférences utilisateurs organisées en catégories), ou une autre relation quelconque (une ressource correspondant à une machine faisant partie d'un réseau, dont les sous-ressources représentent les machines auxquelles elle est connectée directement).

La figure 5.1 représente un exemple d'une partie d'un modèle contextuel modélisant les ressources physiques d'une machine (domaine contextuel sys).

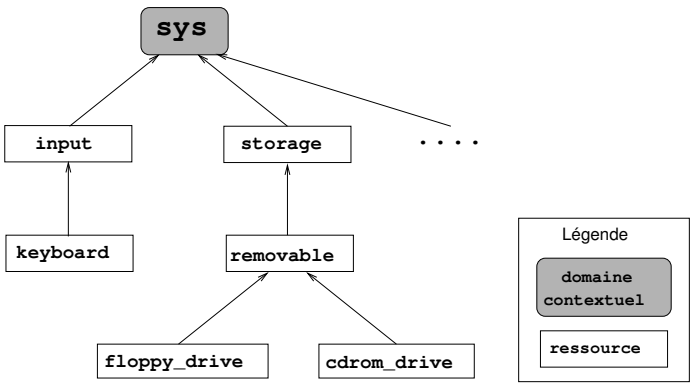


FIG. 5.1 – Exemple de hiérarchie de ressources pour le domaine sys.

WildCAT ne fait aucune différence entre ces différentes utilisations possibles des mécanismes qu'il fournit ; la sémantique associée par l'utilisateur aux différents noeuds de l'arborescence et à leurs relations ne le concerne pas.

Ce choix d'une structure arborescente simple plutôt que, par exemple, d'un graphe comme dans [Henricksen et al., 2002], a été fait car il nous semble un bon compromis entre les critères de simplicité et

de pouvoir de modélisation. L'expérience a montré que ce genre de modèle se prête bien à la modélisation de beaucoup de domaines : tous les systèmes de fichiers depuis UNIX, les annuaires tels LDAP[4], les systèmes de nommage (JNDI[5]), le modèle sous-jacent à XML, etc. De plus, ce genre de modèle peut s'implémenter de façon simple et surtout **efficace**, et tous les programmeurs sont parfaitement familiers avec son utilisation.

Notons que l'absence de certaines fonctionnalités au niveau du modèle, comme par exemple le partage de sous-arbres, qui revient à utiliser un modèle de graphe orienté acyclique (DAG[6]) plutôt qu'un simple arbre, n'empêche pas certaines implémentations de supporter ces fonctionnalités. La seule contrainte est que, vu de l'extérieur, les données se conforment au modèle hiérarchique simple de WildCAT.

Ainsi, quasiment tout ce qui peut être représenté dans un modèle plus complexe comme un graphe peut aussi l'être dans WildCAT, en utilisant des conventions de nommage ou d'organisation. Par exemple, si on veut distinguer les sous-éléments d'une ressource (au sens d'éléments contenus *dans* la ressource) de ces simples voisins (éléments accessibles), on peut utiliser deux sous-ressources « abstraites » fils et voisins qui contiendront respectivement les ressources représentant fils et les voisins. Ces deux ressources intermédiaires servent alors en quelque sorte à qualifier la relation de la ressource parente avec ses sous-ressources. La différence essentielle par rapport à des systèmes supportant directement ce genre de relations « typées », est que WildCAT n'étant pas « conscient » de ces aspects, c'est aux programmeurs d'utiliser le système de façon cohérente.

De ce point de vue, WildCATsacrifie la **richesse du modèle** au profit de la **simplicité d'utilisation** et de la **généralité**, en n'incluant que le minimum nécessaire à toutes les applications. Celles qui ont besoin d'un modèle de données plus riche peuvent le simuler par les moyens décrits ci-dessus, sans que les autres aient à souffrir d'une lourdeur imposée et inutile dans leur cas.

### Description des ressources par des attributs

Chaque ressource d'une arborescence représente un élément (concret ou non) du contexte de l'application. Afin de décrire les caractéristiques de ces éléments, à chaque ressource peut être associé un nombre quelconque d'*attributs*. Un attribut est un simple couple $(nom, valeur)$, les noms des attributs d'une ressource étant uniques pour cette ressource.

Ainsi, une ressource mouse pourrait avoir, entre autres, un attribut buttons dont la valeur indique le nombre de boutons de la souris (voir figure 5.2).

| mouse | |
|---|---|
| brand | "Logitech" |
| model | "MX900" |
| interface | "usb" |
| dpi | 1000 |
| buttons | 6 |
| wheel | true |

FIG. 5.2 – Une ressource mouse représentant une souris, décrite par des attributs.

Les valeurs des attributs sont typées (mais pas les attributs eux-mêmes), mais le modèle ne fait aucune supposition sur les types disponibles. Actuellement, l'implémentation supporte les types de données suivants :

- booléens (vrai ou faux) ;
- nombres (équivalents aux `double` de Java) ;
- chaînes de caractères (équivalentes aux chaînes de Java).

Même si cette liste est limitée, elle s'avère suffisante en pratique dans un grand nombre de cas. Si l'on a besoin de données plus structurées, il est possible d'utiliser des sous-ressources et leurs attributs pour

[4]Lightweight Directory Access Protocol
[5]Java Naming and Directory Interface
[6]Directed Acyclic Graph

les modéliser, où d'étendre l'implémentation (qui est conçue pour cela) pour ajouter de nouveaux types de valeurs ou pour enrichir les types existants[7].

La figure 5.3 représente le modèle de données statique de WildCAT tel qu'il vient d'être décrit, sous la forme d'un diagramme de classes UML. Notons qu'il s'agit d'un modèle *logique*, et qu'il n'implique absolument pas qu'une implémentation doivent fournir les classes correspondantes. En effet, suivant le domaine contextuel, les techniques d'implémentations peuvent beaucoup varier, et une implémentation naïve suivant ce diagramme UML risquerait d'être prohibitive. Par exemple, si l'on veut fournir un domaine contextuel représentant le système de fichiers local, pour permettre aux applications de réagir à la présence ou à l'absence d'un fichier par exemple, une implémentation naïve nécessiterait de créer des dizaines de milliers d'objets en mémoire, pour représenter tous les fichiers présents sur une machine moderne. Comme nous le verrons par la suite (Section 5.4), l'interface de programmation grâce à laquelle les applications peuvent accéder aux informations du contexte a été conçue spécifiquement en ne tenant compte que du modèle de données logique, sans faire de supposition sur l'implémentation. De plus, si WildCAT fournit une implémentation par défaut, le système permet à chaque domaine contextuel d'utiliser une implémentation différente, et donc adaptée à sa problématique.

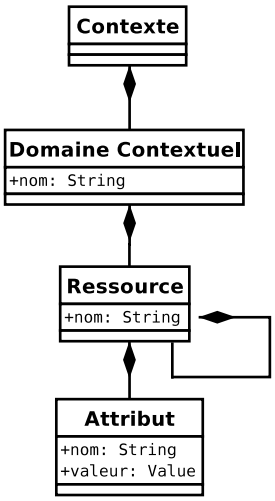


FIG. 5.3 – Modèle de données logique de WildCAT

### 5.3.2 Évolutions dynamiques du contexte

La section précédente décrivait le modèle de données statique utilisé pour représenter les informations contextuelles. Or, le contexte d'exécution d'une application est en général extrêmement dynamique. WildCAT prend en compte cet aspect en permettant au modèle d'évoluer au cours du temps, aussi bien au niveau des valeurs des attributs que de la structure même du modèle. Concrètement, les modifications qui peuvent se produire sont les suivantes :

- Modification des valeurs des attributs. Exemple : la valeur de l'attribut free de la ressource memory évoluera en fonction de la quantité de mémoire disponible.
- Apparition ou disparition d'attributs au sein d'une ressource. Exemple : une ressource environment représentant sous forme d'attributs les variables d'environnement du processus courant peut voir sa liste d'attributs évoluer.
- Apparition ou disparition de ressources (y compris tous leurs attributs et sous-ressources). Exemple : connexion ou déconnexion d'un périphérique tel qu'une clé USB.

[7]Par exemple, l'implémentation actuelle ne supporte pas un certain nombre de méta-données comme la précision ou la confiance que l'on peut accorder à une valeur mesurée [Henricksen et al., 2002].

Pour permettre aux applications de réagir aux évolutions du modèle, qui correspondent à des modifications de leur contexte d'exécution, chaque modification génère un événement spécifique, accompagné de toutes les informations nécessaires pour décrire la modification[8].

Lorsqu'une ressource est créée, le système génère tout d'abord un événement correspondant à la création de la ressource elle-même. Puis, si elle possède des attributs initiaux, un événement est généré pour l'apparition de chacun d'entre eux (dans un ordre non défini). Enfin, si la nouvelle ressource possède des sous-ressources dès sa création, celles-ci sont ensuite créées une par une en suivant le même processus.

À l'inverse, lors de la disparition d'une ressource, le système génère d'abord l'ensemble des événements correspondant à la disparition de ses sous-ressources, puis de ces attributs propres, et enfin de la ressource elle-même.

### 5.3.3 Conclusion

Le modèle de données décrit dans cette section est relativement simple et **moins riche** que d'autres propositions [Henricksen et al., 2002]. Nous avons fait ce choix afin de conserver sa **généralité**, et de permettre une implémentation **efficace**. De plus, comme nous l'avons expliqué plus haut, ce modèle hiérarchique est très familier des programmeurs. Or, étant donné le caractère général de WildCAT, les programmeurs doivent non seulement *interagir* avec le système grâce à des interfaces de programmation (sujet de la section suivante), mais doivent aussi le *configurer* pour leur propre application, voire éventuellement l'*étendre*. Par rapport aux critères définis précédemment, nous avons donc privilégié la **simplicité** d'utilisation (sous toutes ses formes) à la **richesse** du modèle.

## 5.4 Interface de programmation

Cette section décrit l'interface de programmation Java utilisable par les applications clientes de WildCAT. Du point de vue de ses utilisateurs, WildCAT présente une interface relativement **simple**, qui donne accès aux deux modes d'utilisation possibles du service – interrogation synchrone et notification asynchrone – sans exposer la complexité du fonctionnement interne.

Cette interface est constituée principalement de deux classes et d'une interface Java :

- Le point central du système – du point de vue des clients – est la classe `Context`[9] ; c'est elle qui fournit aux clients les services de haut niveau dont ils ont besoin (interrogation du contexte et enregistrement pour notifications ultérieures).
- La classe `Path` est utilisée pour désigner un ou plusieurs élément(s) du contexte (ressource ou attribut) en spécifiant sa localisation dans l'arborescence, et donc de façon indépendante de l'implémentation réelle du contexte.
- Enfin, l'interface `ContextListener` doit être implémentée par les clients qui veulent être notifiés des évolutions du contexte. Elle définit un certain nombre de méthodes de rappel (*callbacks*) correspondant aux différents types d'événements qui peuvent de produire.

Dans cette section, nous ne traitons pas de l'instanciation et de la configuration de la classe `Context` (voir pour cela la section 5.5). Nous supposons que le programme client a déjà obtenu une référence vers une instance déjà configurée de cette classe qui représente son contexte d'exécution et décrivons donc tout d'abord l'interface principale que les programmeurs clients utiliseront le plus.

### 5.4.1 Désignation des ressources et des attributs

La classe `Path` permet aux clients de désigner un ou plusieurs éléments du contexte en indiquant leur localisation dans l'arborescence de ressources et d'attributs qui représentent celui-ci. Un chemin est

[8] Pour des raisons de performance, les implémentations du modèle de donnée de WildCAT ne sont pas obligées de générer *tous* les événements. Cependant, vues de l'extérieur, elles doivent se comporter comme si tous les événements étaient effectivement générés.

[9] Toutes les classes et interfaces mentionnées dans cette section font partie du paquetage `org.obasco.wildcat`. Les noms courts des classes sont utilisés dans le corps du texte pour ne pas le surcharger.

constitué de plusieurs parties :

- un nom de domaine contextuel ;
- une suite de noms de ressources, correspondant au chemin à parcourir depuis la racine du domaine contextuel ;
- éventuellement un nom d'attribut, si le chemin désigne un attribut.

Le nom de domaine contextuel est obligatoire. Les noms des ressources et le nom de l'attribut sont optionnels, mais si un nom d'attribut est présent, le chemin doit comporter au moins un nom de ressource. Autrement dit, le domaine contextuel ne peut pas avoir d'attributs directs. L'élément (ressource ou attribut) désigné par un chemin est défini par rapport au modèle logique de WildCAT, et est totalement indépendant de l'implémentation concrète du domaine contextuel correspondant.

La syntaxe reconnue par la classe `Path` pour créer des chemins est similaire à celle d'une URI : `domaine://chemin/vers/une/ressource@attribut`. La figure 5.4 présente la syntaxe formelle reconnue par `Path`.

```
Path       ::=  Domain "://" Steps
Steps      ::=  Resource ("/" Resource)* Ending?
Ending     ::=  "/*" | "@*" | ("@" Attribute)
Domain     ::=  identifier
Resource   ::=  identifier
Attribute  ::=  identifier
```

FIG. 5.4 – Syntaxe des chemins WildCAT

Le dernier élément du chemin, s'il s'agit d'un nom de ressource (resp. d'attribut), peut être remplacé par le caractère spécial `*`. Le chemin résultant désigne alors toutes les ressources (resp. tous les attributs) localisés à l'emplacement ainsi défini, quel que soit son nom. Par exemple, le chemin `sys://storage/removable/*` désigne toutes les sous-ressources directes de `sys://storage/removable`, alors que `sys://storage/removable/floppy@*` désigne tous les attributs de la ressource floppy.

Quelques exemples de chemins possibles :

`geo://` : la racine du domaine contextuel geo, décrivant le contexte géographique ;

`user://capabilities/vision` : une ressource abstraite décrivant les capacités (ou incapacités) visuelles de l'utilisateur.

`file://usr/bin/*` : toutes les sous-ressources de la ressource bin, représentant les fichiers (et sous-répertoires) du répertoire correspondant ;

`sys://devices/usb/mouse@delta_x` : l'attribut delta_x de la ressource mouse, représentant le déplacement horizontal de la souris ;

`sys://devices/usb/mouse@*` : tous les attributs de la ressource mouse.

La construction d'objets `Path` peut se faire de deux façons :

- en utilisant directement la syntaxe décrite ci-dessus : `new Path("sys ://output/screen")` ;
- à partir d'un chemin existant, qui doit désigner une unique ressource, et que l'on peut étendre avec un nom de sous-ressource et/ou d'attribut : `new Path(parentPath, "subResource", "attr")`. Le deuxième ou le troisième paramètre peuvent être `null`, mais pas les deux.
  Ainsi, si la variable `usb` a pour valeur le chemin `sys://devices/usb`, alors :
  - `new Path(usb, "audio", null)` étend `usb` avec une sous-ressource, et renvoie une valeur désignant le chemin `sys://devices/usb/audio` ;
  - `new Path(usb, null, "version")` étend `usb` avec un attribut, et renvoie une valeur désignant le chemin `sys://devices/usb@version` (représentant la version de la norme USB supportée par le système) ;

### 5.4.2 Interrogation du contexte et navigation

La première interface supportée par la classe `Context` permet aux applications de découvrir la structure du contexte et les valeurs des attributs. Elle est constituée des méthodes suivantes :

`getDomains()` Renvoie sous forme de tableau les noms de tous les domaines contextuels existants dans le contexte au moment de l'appel.

`hasDomain(String name)` Teste l'existence d'un domaine contextuel désigné par son nom.

`exists(Path path)` Teste l'existence d'un chemin donné, qui peut désigner un domaine contextuel, une ressource ou un attribut. Si le chemin désigne un ensemble de ressources (resp. d'attributs), teste l'existence d'*au moins* une ressource (resp. un attribut) à l'endroit désigné.

`resolve(Path attr)` Renvoie la valeur actuelle de l'attribut (unique) désigné par le chemin `attr`. Si l'attribut en question n'existe pas, renvoie `null`. Lève une exception `IllegalArgumentException` si `attr` ne désigne pas un attribut unique.

`getChildren(Path res)` Renvoie la liste des sous-ressources directes de la ressource ou du domaine désignée par `res`, sous-forme des chemins correspondants. Si la ressource existe mais n'a pas de sous-ressources, renvoie un tableau vide. Si la ressource n'existe pas, renvoie `null`. Si `res` désigne un ensemble de ressources, renvoie l'union de toutes leurs sous-ressources. Enfin, si `res` désigne un attribut ou un ensemble d'attributs, lève une exception `IllegalArgumentException`.

`getAttributes(Path res)` Renvoie la liste des attributs de la ressource désignée par `res`, sous-forme des chemins correspondants. Si la ressource existe mais n'a pas d'attributs, renvoie un tableau vide. Si la ressource n'existe pas, renvoie `null`. Si `res` désigne un ensemble de ressources, renvoie l'union de tous leurs attributs. Enfin, si `res` ne désigne pas une ressource unique, lève une exception `IllegalArgumentException`.

Par rapport aux trois besoins principaux identifiés au début du chapitre, les méthodes `getDomains()`, `getChildren()` et `getAttibutes()` permettent la *découverte* de la structure du contexte, alors que `exists()` et `resolve()` permettent son *interrogation*. Les méthodes correspondant au dernier besoin, *enregistrement et notification* seront décrites dans la sous-section suivante.

L'ensemble de ces méthodes permet de découvrir à la fois la structure du contexte et les valeurs des attributs qui décrivent les ressources, qui couvrent la totalité des informations contenues dans le contexte. Les informations renvoyées sont valides au moment de la requête, mais étant donné le caractère dynamique du contexte, il est possible qu'entre le moment où le client reçoit une réponse – par exemple une liste d'attributs – et le moment où il demande plus d'information sur cette réponse, ces informations ne soient plus valides – l'attribut peut avoir disparu au moment où le client en demande la valeur.

Une interface synchrone de ce type ne peut pas éviter ce genre de problèmes, à moins de bloquer temporairement les modifications du modèle lorsque le client l'interroge. Cependant, une telle solution est très difficile à implémenter de façon efficace, nécessite que le client indique toujours la fin d'une séquence d'appels (le système ne peut pas la prévoir) et n'est de toute façon pas conforme à la réalité que le modèle est censé représenter. L'interface asynchrone décrite dans la section suivante permet de résoudre ces problèmes en tenant mieux compte de l'aspect dynamique du contexte.

### 5.4.3 Abonnement et notifications asynchrones

La seconde interface fournie par la classe `Context` permet aux clients de WildCAT de s'enregistrer auprès de celui-ci pour être notifiés de façon asynchrone lorsque certains événements se produisent dans le contexte.

Dans un premier temps, un client doit s'enregistrer en spécifiant d'une part quelles sont les circonstances spécifiques qui l'intéressent et d'autre part un destinataire qui recevra les notifications. À partir de cet instant, à chaque fois que le service d'observation détecte une modification du contexte qui correspond aux attentes d'un client enregistré, il notifie celui-ci en invoquant une méthode spécifique du destinataire, auquel sont passées toutes les informations nécessaires pour caractériser l'événement. Bien entendu, le client peut à tout moment se désabonner pour ne plus être notifié. La figure 5.5 illustre ce protocole. Le système garantit que les notifications d'événements envoyées à un abonné se produisent dans le même ordre que les événéments eux-mêmes ; cependant, aucune garantie n'est faite concernant les délais de notifications, même si le système fait de son mieux pour qu'ils soient les plus courts possibles (*best-effort*).

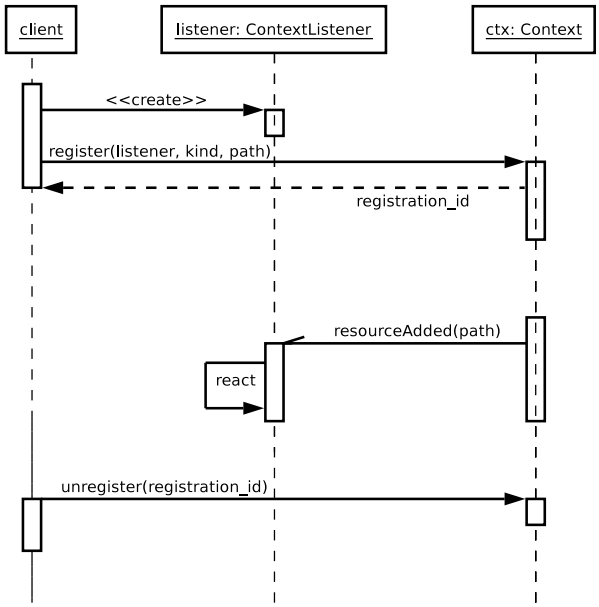


FIG. 5.5 – Protocole d'enregistrement / notification de WildCAT.

Comme on l'a vu précédemment, lors des évolutions dynamiques du contexte, cinq types d'événements peuvent se produire :

1. l'apparition d'une nouvelle ressource ;
2. la disparition d'une ressource ;
3. l'apparition d'un nouvel attribut ;
4. la disparition d'un attribut ;
5. et enfin le changement de valeur d'un attribut.

Pour permettre au client d'indiquer le ou les types d'événements auquel(s) il s'intéresse, WildCAT fournit l'interface `EventKind` (cf. figure. 5.6), qui définit une constante pour chacun des types possibles. En plus des événements correspondant directement aux modifications du modèle, `EventKind` définit deux autres constantes, désignant deux types d'événements plus abstraits, décrits plus loin.

```
public interface EventKind {
    int RESOURCE_ADDED     = 0x01;
    int RESOURCE_REMOVED   = 0x01 << 1;
    int ATTRIBUTE_ADDED    = 0x01 << 2;
    int ATTRIBUTE_REMOVED  = 0x01 << 3;
    int ATTRIBUTE_CHANGED  = 0x01 << 4;
    int EXPRESSION_CHANGED = 0x01 << 5;
    int CONDITION_OCCURED  = 0x01 << 6;
}
```

FIG. 5.6 – Les différents types d'événements définis par l'interface `EventKind`.

L'utilisation des constantes entières pour désigner ces événements permet de combiner plusieurs constantes à l'aide des opérateurs de manipulation de bits de Java. Par exemple, si un client est intéressé aussi bien par l'apparition que par la disparition de ressources sur un chemin donné, il peut

utiliser l'expression Java `RESOURCE_ADDED | RESOURCE_REMOVED` pour indiquer les types d'événements qui l'intéressent, sans avoir à s'abonner à chacun des types individuels.

L'abonnement se fait simplement en invoquant l'une des deux méthodes `register()` de la classe `Context` (cf. figure 5.7). La seule différence entre les deux méthodes est que la première est utilisée lorsque le client est intéressé par les modifications se produisant sur un chemin particulier, alors que la seconde, plus générale, permet d'être notifié des évolutions d'une expression quelconque concernant le contexte de l'application. Ainsi, la première méthode correspond aux cinq premiers types d'événements décrits par `EventKind`, et la seconde aux deux derniers.

```
public class Context {
    ...
    public long register(ContextListener listener, int eventKinds, Path path) { ... }
    public long register(ContextListener listener, int eventKind, String expr) { ... }
    public void unregister(long regId) { ... }
    ...
}
```

FIG. 5.7 – L'interface d'abonnement de la classe `Context`.

Les paramètres de ces méthodes sont :

1. `listener` : Un objet implémentant l'interface `ContextListener`, qui recevra les notifications lors de l'occurrence des événements.
2. `eventKinds` : Un masque de bits représentant le ou les types d'événements auxquels le client s'intéresse.
3. Suivant la méthode utilisée :
   - `path` : Le chemin auquel le client s'intéresse, qui peut représenter éventuellement un ensemble de ressources ou d'attributs (en utilisant `*`), mais qui doit être compatible avec le masque d'événements. Si ce n'est pas le cas, une exception `IllegalArgumentException` est levée.
   - `expr` : Une expression concernant le contexte, dans un langage simple qui sera décrit plus loin. Dans le cas de cette méthode, le masque d'événement `eventKinds` ne peut contenir que les événements `EXPRESSION_CHANGED` et `CONDITION_OCCURED` (éventuellement les deux). Sinon, une exception `IllegalArgumentException` est levée.

Dans tous les cas, les méthodes renvoient un identifiant unique (*cookie*) sous la forme d'un nombre de type `long`, qui représente cet abonnement particulier. Cet identifiant doit être conservé par le client car il est nécessaire pour se désabonner. Lorsque le client n'est plus intéressé par un certain événement, il lui suffit d'invoquer la méthode `unregister()` de la classe `Context`, en passant cet identifiant en paramètre.

Une fois le client abonné, le service le préviendra à chaque fois qu'un événement correspondant à sa demande se produira dans le contexte, en invoquant la méthode appropriée du `ContextListener` indiqué lors de l'abonnement. La notification se fait toujours *après* que l'événement correspondant ait eu lieu. Dans le cas de la première méthode, les événements correspondent à des modifications de la structure du contexte ou des valeurs des attributs. Le fonctionnement de la seconde méthode, qui spécifie une expression arbitraire, est plus complexe :

- le service d'observation recalcule automatiquement la valeur de l'expression à chaque fois que l'un des éléments du contexte qu'elle mentionne est modifié[10] (par exemple la valeur d'un attribut) ;
- si le client avait spécifié `EXPRESSION_CHANGED` comme type d'événement, alors il est notifié si la valeur globale de l'expression est modifiée d'une quelconque façon ;
- si le client avait spécifié `CONDITION_OCCURED` comme type d'événement, alors il n'est notifié que si la valeur globale de l'expression, considérée en tant que valeur booléenne, passe de faux (`false`) à vrai (`true`).

[10] L'implémentation actuelle recalcule l'intégralité de l'expression à chaque fois, ce qui peut parfois poser des problèmes de performances. Une des améliorations futures de WildCAT pourrait consister à effectuer ce calcul de façon incrémentale.

Ainsi, plutôt que d'être notifié à chaque fois que la valeur d'un attribut change, il est par exemple possible à un client de n'être notifié que s'il passe en dessous d'un certain seuil, en invoquant une commande telle que :

```
context.register(aListener, EventKind.CONDITION_OCCURED, "sys://storage/memory.free < 15000");
```

En théorie, il est bien sûr possible à un client d'obtenir le même résultat en s'abonnant aux modifications de l'attribut, puis en effectuant lui-même le test lors de chaque notification. Cependant, La solution supportée par WildCAT rend l'utilisation du système beaucoup plus simple, et surtout évite d'effectuer un grand nombre de notifications inutiles qui risqueraient de ralentir l'application. En rendant le système d'observation capable de prendre des décisions locales plus complexes, les applications clientes ne sont notifiées que des circonstances qui demandent réellement une réaction de leur part.

Chaque type d'événement définit par `EventKind` correspond à une méthode spécifique de l'interface `ContextListener` (cf. figure 5.8), qui doit être implémentée par le destinataire des notifications. La classe `ContextListenerAdapter` fournit une implémentation vide de cette interface, qui permet de rapidement créer des sous-classes ne gérant qu'un type d'événement (typiquement dans une classe interne anonyme).

```
public interface ContextListener {
    void attributeAdded(Path attr, long timeStamp);
    void attributeRemoved(Path attr, long timeStamp);
    void attributeChanged(Path attr, Value oldVal, Value newVal, long timeStamp);
    void resourceAdded(Path res, long timeStamp);
    void resourceRemoved(Path res, long timeStamp);
    void expressionValueChanged(long regId, Value oldVal, Value newVal, long timeStamp);
    void conditionOccured(long regId, long timeStamp);
}
```

FIG. 5.8 – L'interface `ContextListener` utilisée pour notifier les clients.

Voici la signification précise de chaque type d'événement, ainsi que la correspondance entre les codes définis par `EventKind` et les méthodes de `ContextListener` :

- `RESOURCE_ADDED` → `resourceAdded(Path res, long ts)`
  Apparition d'une ressource dans le contexte, dont le chemin est `res`, à l'instant désigné par `ts` (correspondant à la valeur renvoyée par `System.currentTimeMillis()`).
- `RESOURCE_REMOVED` → `resourceRemoved(Path res, long ts)`
  Disparition d'une ressource dans le contexte, dont le chemin était `res`, à l'instant désigné par `ts`.
- `ATTRIBUTE_ADDED` → `attributeAdded(Path attr, long ts)`
  Apparition d'un attribut dans le contexte, dont le chemin est `attr`, à l'instant désigné par `ts`.
- `ATTRIBUTE_REMOVED` → `attributeRemoved(Path attr, long ts)`
  Disparition d'un attribut dans le contexte, dont le chemin était `attr`, à l'instant désigné par `ts`.
- `ATTRIBUTE_CHANGED` → `attributeChanged(Path attr, Value oldVal, Value newVal, long ts)`
  Modification de la valeur de l'attribut désigné par `attr`, passant de l'ancienne valeur `oldVal` à la nouvelle `newVal`, à l'instant `ts`.
- `EXPRESSION_CHANGED` → `expressionChanged(long regId, Value oldVal, Value newVal, long ts)`
  Modification de la valeur d'une expression. `regId` est l'identifiant de la requête renvoyée au client lorsqu'il s'est enregistré pour connaître les évolutions de l'expression. La valeur de l'expression en question est passée de `oldVal` à `newVal` à l'instant `ts`.
- `CONDITION_OCCURED` → `conditionOccured(long regId, long ts)`
  Occurrence d'une condition, c'est-à-dire passage de la valeur d'une expression – interprétée comme un booléen – de faux à vrai. `regId` est l'identifiant de la requête renvoyée au client lorsqu'il s'est enregistré pour connaître les occurrences de la condition. La valeur de la condition en question est passée de faux à vrai à l'instant `ts`.

## 5.5 Instanciation et configuration du système

Les sections précédentes présentaient l'utilisation de WildCAT du point de vue des programmeurs clients utilisant le service dans leur application, mais ne décrivaient pas comment initialiser et configurer le système. Puisque WildCAT est un système générique, cette section décrit comment chaque application doit configurer le système en spécifiant quels sont les éléments qui constituent son contexte d'exécution, et comment les observer.

La classe `Context` s'instancie très simplement en faisant appel à son unique constructeur, qui ne prend pas de paramètres. Une instance de cette classe nouvellement créée représente initialement un contexte « vide », c'est-à-dire qui ne contient aucun domaine contextuel. Typiquement, une application qui veut utiliser les services de WildCAT crée une unique instance de cette classe lors de son initialisation, puis la configure de façon spécifique à son contexte d'exécution propre.

Configurer cet objet revient à y ajouter un ou plusieurs domaines contextuels, en utilisant tout simplement la méthode `Context.addDomain(aDomain)`. L'unique paramètre de cette méthode doit être une instance de l'interface `ContextDomain`.

Comme nous l'avons expliqué plus haut, cette conception permet à chaque domaine contextuel d'avoir une implémentation différente et spécifique, et ainsi d'être le plus efficace possible. La responsabilité de la classe `Context` se limite en fait à servir de *façade* [Gamma et al., 1994] à l'ensemble des domaines contextuels disponibles, et à gérer les interactions entre ceux-ci ainsi que les événements spéciaux liés aux évolutions d'expressions (`EventKind.EXPRESSION_CHANGED` et `EventKind.CONDITION_OCCURED`).

WildCAT fournit une implémentation standard de l'interface `ContextDomain` sous la forme de la classe `StandardDomain` du paquetage `org.obasco.wildcat.domain.standard`[11]. Cette implémentation se veut à la fois simple, générique, et relativement efficace. Elle est basée sur une implémentation directe du modèle de données logique de WildCAT (cf. figure 5.3, page 78). Cette implémentation générique peut être configurée par l'intermédiaire d'un fichier XML dont le format est décrit dans la suite de cette section. Utiliser cette implémentation se fait très simplement :

```
Context ctx = new Context();
File config = new File("définition-domaine.xml");
ctx.addDomain(new StandardDomain("nom-domaine", config));
```

Le fichier XML utilisé pour configurer l'implémentation standard d'un domaine contextuel décrit :
- la structure initiale du domaine contextuel, c'est-à-dire les ressources et attributs qui existent dès sa création ;
- la définition des attributs de chacune de ces ressources ;
- les règles gouvernant les évolutions dynamiques de la structure du domaine contextuel.

### 5.5.1 Définition de la structure statique

La définition de la structure statique initiale du domaine contextuel se fait très simplement, en utilisant l'élément XML `resource`, le nom de la ressource étant spécifié par l'attribut `name`. L'imbrication des éléments XML correspond à l'imbrication des ressources du domaine. Le domaine contextuel lui-même est représenté par l'élément racine `context-domain`. Voici un exemple simple qui décrit un domaine contextuel entièrement statique, et constitué uniquement de ressources (sans attributs) :

```
<?xml version='1.0' encoding='ISO-8859-15'?>
<context-domain>
  <resource name="eth0">
    <resource name="inet-config" />
    <resource name="traffic" />
      <resource name="input" />
```

[11] Toutes les implémentations de domaines contextuels fournies avec WildCAT se trouvent dans un sous-paquetage approprié de `org.obasco.wildcat.domain`.

```
      <resource name="output" />
    </resource>
  </resource>
  <resource name="lo" />
</context-domain>
```

Concernant les attributs, on en distingue deux types :

1. les attributs primitifs, correspondant directement à une valeur observée dans le contexte d'exécution ;
2. les attributs synthétiques, définis en fonction d'autres attributs et/ou ressources par l'intermédiaire d'une expression.

### 5.5.2 Définition des sondes pour les attributs primitifs

L'observation du contexte se fait par l'intermédiaire de *sondes*, de simples objets Java implémentant une interface particulière (l'implémentation des sondes sera développée dans la section 5.6.1). Chaque sonde est chargée d'observer un élément particulier du contexte, et d'envoyer au système les informations concernant cet élément à chaque fois que celles-ci changent. Ces informations sont transmises sous la forme d'un ensemble de couples de mesures $(nom, valeur)$ décrivant les propriétés de l'élément observé.

À chaque ressource décrite dans le fichier de configuration XML, il est possible d'attacher une ou plusieurs sondes, chacune ayant un nom unique. Lorsqu'une sonde est attachée à une ressource, le système considère que les propriétés mesurées par la sonde correspondent à des attributs de la ressource. Ainsi, à chaque fois qu'un nouvel ensemble de mesures est envoyé au système par la sonde, WildCAT modifie les valeurs les attributs correspondants, crée de nouveaux attributs si la sonde envoie de nouveaux noms, et supprime les attributs correspondant aux mesures qui n'apparaissent plus. Afin d'éviter les conflits entre les noms d'attributs si plusieurs sondes sont définies dans une même ressource, les noms des mesures renvoyés par les sondes sont précédées du nom de la sonde et d'un point afin de déterminer le nom de l'attribut correspondant. Si une sonde nommée `s` renvoie une mesure $(m, v)$, l'attribut correspondant sera ainsi nommé `s.m`.

La définition d'une sonde se fait très simplement en utilisant l'élément XML `sensor`. Cet élément possède deux attributs obligatoires :

- `name` : le nom local de la sonde ;
- `class` : le nom complet de la classe Java qui implémente cette sonde.

La définition d'une sonde peut contenir deux sous-éléments, l'un servant à configurer la sonde (`configuration`), et l'autre indiquant les paramètres d'ordonnancement qui détermine quand la sonde doit effectuer ses mesures (`schedule`).

L'élément `configuration` est destiné à contenir des informations de configuration spécifiques à la sonde. Cela permet de créer par exemple une classe unique chargée de récupérer les informations concernant un système de fichier (point de montage, type, taille, place restante...), et un instance de cette classe pour chaque système de fichier monté, en lui indiquant lequel observer grâce à ces informations de configuration. Le contenu de l'élément `configuration` peut être n'importe quel fragment XML bien formé, et n'est pas interprété par WildCAT. Ce fragment est passé tel quel à la sonde lors de son initialisation, sous la forme d'un objet `org.jdom.Element`[12].

L'élément `schedule` n'est obligatoire que pour certains types de sondes, dites passives, et indique au système WildCAT quand effectuer les mesures sur cette sonde. Les types d'ordonnancement supportés actuellement sont :

- `on-create` : indique que la sonde doit être consultée une seule fois, juste après son initialisation. Cet élément convient pour les éléments du contexte qui n'évoluent pas, comme par exemple le microprocesseur. Pour indiquer ce type d'ordonnancement, il suffit d'utiliser un élément XML `on-create` vide :

[12] WildCAT utilise en interne la bibliothèque JDOM pour analyser les fichiers XML (`http ://jdom.org/`).

```
<schedule> <on-create /> </schedule>
```

– `absolute` : indique que la sonde doit être consultée une seule fois, à un instant précis, spécifié de façon absolue.

```
<schedule> <absolute date="yyyy-mm-ddThh:mm:ss" /> </schedule>
```

– `periodic` : indique que la sonde doit être consultée régulièrement, toutes les $n$ millisecondes. Deux attributs optionnels permettent de spécifier l'instant de la première consultation et une limite après laquelle la sonde ne sera plus consultée.

```
<schedule>
  <periodic period="n" [from="yyyy-mm-ddThh:mm:ss"]
                       [until="yyyy-mm-ddThh:mm:ss"] />
</schedule>
```

Voici quelques exemples de définitions de sondes :

```
<?xml version='1.0' encoding='ISO-8859-15'?>
<context-domain>
  <resource name="load">
    <sensor name="load" class="org.obasco.wildcat.sensors.LoadSensor">
      <schedule><periodic period="10000"/></schedule>
    </sensor>
  </resource>
  <resource name="storage">
    <resource name="volatile">
      <resource name="memory">
        <sensor name="memory" class="org.obasco.wildcat.sensors.MemorySensor">
          <schedule><periodic period="2000"/></schedule>
        </sensor>
      </resource>
    </resource>
    <resource name="permanent">
      <sensor name="hda" class="org.obasco.wildcat.sensors.HardDriveSensor">
        <schedule><on-create/></schedule>
        <configuration>
          <device>/dev/hda</device>
          <mode>static</mode>
        </configuration>
      </sensor>
    </resource>
  </resource>
</context-domain>
```

Ce domaine contextuel définit une ressource load décrivant la charge du système, qui aura comme attributs les mesures effectuées toutes les 10 secondes par une instance de la classe `LoadSensor`. Il définit aussi une ressource storage/volatile/memory, mise à jour toutes les deux secondes, et une ressource storage/permanent/hda. Cette dernière n'est observée qu'une seule fois, lors de son initialisation. Les informations de configuration passées à la classe `HardDriveSensor` lui indiquent quel périphérique observer, et lui disent de ne pas effectuer de mesures dynamiques. En effet, cette sonde, par défaut en mode `dynamic`, peut aussi renvoyer des informations sur les performances du disque (taux de transfert...), mais cela n'a évidemment pas de sens si on ne l'invoque qu'une seule fois.

### 5.5.3 Définition d'attributs synthétiques

Les attributs primitifs définis dans la section précédente permettent d'obtenir des informations sur le contexte d'exécution, mais ces informations restent d'un niveau d'abstraction très faible. En effet, les

sondes sont conçues pour obtenir des informations brutes, sans les interpréter. Le plus souvent, les informations fournies, par exemple par le système d'exploitation ou directement par le matériel, sont de très bas niveau. Par exemple, dans le cas particulier de l'observation du trafic réseau, on peut très bien imaginer des sondes indiquant le nombre d'octets entrant ou sortant par une interface réseau, ces informations étant relativement faciles à obtenir du système d'exploitation. Du point de vue de l'application à adapter, il est cependant difficile de raisonner directement à partir de ces données de très bas niveau.

En plus des attributs primitifs présentés jusqu'ici (et qui correspondent directement à une valeur observée par une sonde), nous proposons donc la notion d'*attributs synthétiques*, définis par une expression pouvant faire référence à n'importe quel autre ressource ou attribut disponible (les définitions récursives sont interdites). Les valeurs de ces attributs synthétiques sont mises à jour automatiquement par le système à chaque fois qu'une de leurs dépendances évolue, exactement de la même manière que les clients peuvent être notifiés des évolutions d'une expression en utilisant `EventKind.EXPRESSION_CHANGED`. Pour reprendre l'exemple précédent, à partir du nombre d'octets entrant et sortant, il est ainsi possible de calculer le débit du réseau en kB/s (ce qui est déjà une information plus facilement utilisable par une application), voire, en combinant ce nouvel attribut avec le débit maximum supporté par l'interface (obtenu par une autre sonde, ou un autre attribut de la même sonde), de déterminer un taux d'utilisation exprimé en pourcentage. À partir des données primitives observées par les sondes, les attributs synthétiques permettent d'obtenir des informations de plus haut niveau, ayant une signification plus facilement interprétable par les applications.

La définition d'attributs synthétiques se fait en utilisant l'élément XML `attribute`. Le nom du nouvel attribut est spécifié par l'attribut XML `name`, et la définition de l'attribut est spécifiée dans le corps de l'élément XML. Le langage utilisé pour exprimer ces expressions est très simple. Il comprend :

- des valeurs littérales pour chacun des types supportés par le système :
  - les booléens : `true` et `false` ;
  - les nombres : `42`, `-1`, `3.14`, `6.02E-23` ;
  - les chaînes de caractères : `"foo"`, `"\"bar\""`.
- la possibilité de référencer des ressources et des attributs, y compris des ensembles de ressources ou d'attributs :
  - par un chemin absolu, en utilisant la syntaxe définie précédemment : `sys://devices/usb/*`, `sys://storage/volatile/memory@free` ;
  - par un chemin relatif à la ressource où l'attribut est défini, en utilisant la convention familière pour représenter la ressource courante (`.`) et son parent (`..`) : `./mouse@model`, `../../input/keyboard@nb\_keys` ;
- la possibilité de référencer des variables, en utilisant la syntaxe `$nom_variable` (le mécanisme utilisé pour définir ces variables est décrit plus loin) ;
- les opérateurs arithmétiques standards `+`, `-`, `*` et `div`[13] avec leur priorités habituelles, et la possibilité de parenthéser les expressions. Ces opérateurs ont la même sémantique qu'en Java lorsqu'ils sont utilisés avec des nombres. Toujours comme en Java, l'opérateur `+` peut être utilisé pour concaténer des chaînes de caractères.
- les opérateurs logiques standards : `or`, `and`, `not` ;
- la possibilité d'invoquer des fonctions : `concat("foo", "bar")`. Toutes les fonctions standards sont sans effet de bords.

Il est ainsi possible de définir de nouveaux attributs qui combinent les valeurs d'autres attributs, qu'ils soient primitifs ou synthétiques, offrant ainsi des informations de plus haut niveau et plus facilement interprétables par les clients du service. Du point de vue des clients, il n'y a aucune différence entre les deux types d'attributs.

Voici quelques exemples d'attributs synthétiques :

```
<resource name="hard-drives">
  <resource name="hda">
    <attribute name="model">"MAXTOR 6L040L2"</attribute>
```

[13] `div` est utilisé à la place de `/` pour la division, pour éviter les conflits avec la syntaxe des chemins.

```
    <attribute name="size_mb">40000000</attribute>
    <attribute name="size_gb">@size_mb div 1024</attribute>
  </resource>
  <resource name="hdb">
    <attribute name="model">"ST3120022A"</attribute>
    <attribute name="size_mb">120000000</attribute>
    <attribute name="size_gb">@size_mb div 1024</attribute>
  </resource>
  <attribute name="total-size_gb">hda@size_bg + hdb@size_gb</attribute>
</resource>

<resource name="network">
  <attribute name="nb_interfaces">count(*)</attributes>
  <attribute name="error_rate">average(*, "error_rate")</attribute>
  <resource name="eth0">
    <sensor name="nic" class="org.obasco.wildcat.sensors.NICSensor">
      <schedule><periodic period="1000"/></schedule>
      <configuration><device>eth0</device</configuration>
    </sensor>
    <attribute name="error_rate">@nic.dropped_packets div @nic.received_packets</attribute>
  </resource>
</resource>
```

### 5.5.4 Modélisation des aspects dynamiques du contexte

Les constructions présentées jusqu'ici ne permettent que de modéliser la structure statique du contexte, et les évolutions des attributs. Elles ne permettent pas de spécifier les évolutions de la structure même du contexte. Pour cela, nous introduisons deux nouvelles constructions basées sur la notion de gabarit (*template*).

#### Branches conditionnelles simples

La première construction permet de définir une branche de l'arborescence dont l'apparition est conditionnée par une expression booléenne. Pour cela, il suffit d'utiliser un élément XML `if` contenant une arborescence de ressources et d'attributs normaux (voire d'autres branches conditionnelles). Cet élément `if` possède un attribut `test` qui définit une expression quelconque du même type que celles utilisées pour définir les attributs synthétiques.

Par exemple :

```
<!-- La valeur booléenne d'une ressource ou d'un attribut est vrai ssi celui-ci existe. -->
<if test="sys://devices/usb/mouse">
  <resource name="mouse">
    ...
  </resource>
</if>
<if test="sys://devices/usb/keyboard or sys://devices/ps2/keyboard">
  <resource name="keyboard">
    ...
  </resource>
</if>
```

La sémantique de cette construction est la suivante. Lorsqu'une telle branche est définie, le système observe les évolutions de l'expression définie par l'attribut `test`.

- Si la valeur de l'expression, convertie en booléen, devient `true` (soit initialement, soit après avoir été `false`), alors le système instancie le gabarit comme si les tags `<if ...>` et `</if>` n'apparaissaient pas dans le texte.

– Si la valeur de l'expression, convertie en un booléen, devient `false`, alors le système supprime du contexte (ou, initialement, ignore) les ressources contenues à l'intérieur, exactement comme si tout l'élément `if` et son contenu n'étaient pas présents.

Bien entendu, à chaque changement, le système génère les événements appropriés pour permettre aux clients de réagir à ces évolutions.

Il est tout à fait possible d'imbriquer les gabarits. Dans ce cas, un gabarit qui est inclus dans un autre ne sera pris en compte par le système que si ce dernier est instancié, et si le gabarit englobant disparaît, tout son contenu disparaît avec lui, qu'il soit statique ou dynamique.

### Branches conditionnelles multiples

La seconde construction dynamique est en fait une généralisation de la première, qui permet un nombre quelconque d'instanciations d'un gabarit paramétré par une variable.

Pour cela, on utilise l'élément XML `foreach`. Celui-ci possède deux attributs :

1. `variable`, qui spécifie un nom de variable, utilisé pour paramétrer le gabarit ;
2. et `in`, qui spécifie une expression dont la valeur doit être un *ensemble* de ressources ou d'attributs.

```
<foreach variable="device" in="sys://devices/usb/*">
  <resource name="...">
    ...
  </resource>
</foreach>
```

La sémantique de `foreach` est similaire à celle de `if`, excepté que le système instancie une branche pour *chaque* élément de l'ensemble retourné par l'expression `in`. Puisque la valeur de cette expression peut varier dynamiquement, de nouvelles branches peuvent apparaître ou d'anciennes branches disparaître en cours d'exécution. Avant d'être instanciée, la branche correspondant à un élément donné est transformée par le système :

– Si l'expression renvoie un ensemble de *ressources*, les occurences de la variable désignée dans l'attribut `variable`, sous la forme `$var`, sont remplacées par le nom de la ressource correspondant à la branche.
– Si l'expression renvoie un ensemble d'*attributs*, les occurences de `$var` sont remplacées par le nom de l'attribut, et les occurences de `$var_value` sont remplacées par sa valeur.

Ces variables peuvent être utilisées soit dans le contenu d'attributs soit à l'intérieur d'éléments XML.

Il est ainsi possible de faire dépendre la *structure* d'une partie du contexte de la *valeur* de certaines expressions (en particulier de certains attributs). Imaginons par exemple qu'une application dépende d'informations concernant les systèmes de fichiers de son hôte. Il n'est pas possible de construire statiquement une description générique de ces informations, car le nombre et le type de disques et de systèmes de fichiers varie d'un système à l'autre, voire varie dynamiquement pour un système donné (branchement à chaud d'un disque dur externe par exemple). Cependant, avec les nouvelles constructions que nous avons introduites, ce problème peut se résoudre de la manière suivante :

1. Le programmeur définit une sonde générique interrogeant le système pour connaître la liste des systèmes de fichiers disponibles[14]. À chaque fois qu'elle est interrogée, cette sonde renvoie une liste de couples $(nom\_partition, point\_montage)$, par exemple (`hda1`, "/"), (`hda2`, "`/home`").
2. Il ajoute ensuite cette sonde à une ressource abstraite filesystems.
3. À l'intérieur de cette ressource, il spécifie un gabarit `foreach` de la façon suivante :

```
<foreach variable="partition" in="@*">
  <resource name="$partition"> <!-- Remplacé par le nom de l'attibut: hda1, hda2... -->
    <sensor name="fs" class="org.obasco.wildcat.sensors.FileSystemSensor">
```

[14] En pratique, ce genre de sonde générique peut être codé une fois pour toute et réutilisé directement par de nombreuses applications.

```
        <configuration>
          <device>$partition</device> <!-- Remplacé par hda1, hda2... -->
        </configuration>
        <schedule><periodic period="10000"/></schedule>
      </sensor>
       <!-- $xxx_value est remplacé par la valeur de l'élément, ici "/", "/home"... -->
      <attribute name="mount_point">$partition_value</attribute>
    </resource>
  </foreach>
```

L'expression utilisée, `.@*` renvoie l'ensemble des attributs de la ressource filesystems, en l'occurrence `@hda1`, `@hda2` (notons qu'il s'agit bien des attributs, pas de leurs valeurs). Une nouvelle branche est donc créée pour chacun de ces éléments, en utilisant le gabarit. Pour chacune de ces branches, la variable `$partition` est substituée par la valeur appropriée :

- Lorsque la variable est utilisée là où une expression est attendue, comme par exemple dans la définition de l'attribut mount_point, la valeur utilisée est directement la valeur de la variable, ici un attribut ; on doit utiliser la fonction `value()` pour récupérer la valeur d'un attribut.
- Lorsque la variable est utilisée en dehors d'un tel contexte, la valeur utilisée est le nom de l'élément. C'est le cas par exemple dans la définition du nom de la ressource à instancier.

Dans notre exemple, le résultat initial serait donc :

```
<resource name="hda1">
  <attribute name="mount_point">"/"</attribute>
  <!-- Plus les attributs spécifiques à la sonde configurée pour
       observer hda1. -->
</resource>
</resource name="hda2">
  <attribute nape="mount_point">"/home"</attribute>
  <!-- Plus les attributs spécifiques à la sonde configurée pour
       observer hda2. -->
</resource>
```

Si maintenant l'utilisateur introduit un CD-ROM, la sonde associée a la ressource filesystems renverra un nouvel attribut (`hdc`, "`/mnt/cdrom`"). Réalisant cela, le système instanciera donc une troisième fois le gabarit pour obtenir :

```
<resource name='hdc'>
  <attribute nape='mount_point'>"/mnt/cdrom"</attribute>
  <!-- Plus les attributs spécifiques à la sonde configurée pour
       observer hdc. -->
</resource>
```

Bien entendu, cette ressource disparaîtra en même temps que l'attribut correspondant, lorsque l'utilisateur éjectera le CD-ROM.

### 5.5.5 Conclusion

Dans cette section, nous avons présenté l'implémentation standard de la notion de domaine contextuel fournie par WildCAT. Cette implémentation est générique et peut être configurée à l'aide d'un fichier XML. La structure de ce fichier correspond à la structure voulue du contexte, exceptée en ce qui concerne les évolutions dynamiques. Pour gérer cet aspect, nous avons introduit deux constructions : l'une permet d'avoir des branches de l'arborescence dont l'existence dans le contexte en conditionnée par une expression ; l'autre – généralisation de la première – utilise une notion de gabarit paramétré par une variable et qui peut être instancié par le système un nombre quelconque de fois, nombre qui peut varier dynamiquement.

## 5.6 Mécanismes d'extensions de WildCAT

L'implémentation par défaut décrite dans la section précédente est très configurable, mais ne fournit qu'un ensemble réduit de sondes en standard, et peut ne pas convenir à certaines applications ou pour certains domaines contextuels. Concrètement, une application qui veut utiliser WildCAT pour observer son contexte d'exécution doit étendre le framework pour ses besoins spécifiques.

Cette section décrit deux possibilités d'extensions, la première, très simple, consistant à développer de nouvelles sondes utilisables avec l'implémentation standard (Section 5.6.1), et la seconde, plus complexe, à ré-implémenter intégralement l'interface `ContextDomain` pour des domaines contextuels spécifiques (Section 5.6.2).

### 5.6.1 Développement de nouvelles sondes

L'implémentation actuelle de WildCAT ne fournit qu'un nombre limité de sondes en standard : utilisation du processeur, de la mémoire, charge système... Rien n'empêche le développement de bibliothèques de sondes génériques et réutilisables ; le système a été conçu spécifiquement pour permettre ce genre de choses. Cependant, tant que ces bibliothèques n'existent pas, les applications qui veulent bénéficier des services de WildCAT doivent créer leurs propres sondes, ce qui, comme nous allons le voir, est finalement assez simple.

Le rôle d'une sonde est d'*acquérir* des données brutes décrivant certains aspects du contexte de l'application, et des rendre ces données utilisables par le reste du système. La technique employée pour l'acquisition dépend bien évidemment de la nature des données, et donc du domaine contextuel. Par exemple, dans le cas des ressources systèmes, l'acquisition pourra se faire relativement simplement si le système d'exploitation sous-jacent fournit les informations requises. À l'inverse, obtenir des informations sur l'environnement physique, comme la température ambiante ou la localisation géographique requiert à la fois du support matériel (thermomètre, GPS) et logiciel. Nous ne nous intéressons ici qu'à la partie logicielle.

Comme nous l'avons indiqué à de nombreuses reprises, l'une des caractéristiques principales du contexte d'exécution est qu'il est extrêmement dynamique. L'acquisition des données par les sondes ne doit donc pas se faire une seule fois, mais doit être répétée plus ou moins régulièrement pour s'assurer que la représentation du contexte par WildCAT soit le plus proche possible de son état réel.

WildCAT distingue deux types de sondes selon la façon dont ces mesures répétées sont effectuées [Schroeder, 1995] :

- les sondes actives, qui sont autonomes une fois créées, et ont leurs propres fil d'exécution (*thread*). Elles correspondent au mode *tracing* décrit dans [Schroeder, 1995].
- les sondes passives, qui n'effectuent leurs observations du système que lorsqu'on le leur demande explicitement. Elles correspondent au mode *sampling* de [Schroeder, 1995].

#### Sondes actives

Les sondes actives sont des objets Java implémentant l'interface `Sensor` (cf. figure 5.9). Elles sont caractérisées par un nom et un état (démarré ou stoppé). À chacune de ces sondes, on peut associer un unique `SamplesListener`, chargé de recevoir les mesures effectuées par la sonde. Lorsque l'état d'une sonde est *démarrée* (i.e. `isStarted()` renvoie `true`), la sonde envoie de nouvelles mesures à son écouteur (si elle en a un) à chaque fois que cela est nécessaire. Puisqu'une telle sonde a son propre fil d'exécution, elle a l'entière responsabilité de décider quand envoyer ces informations. L'écouteur qui lui est associé doit être préparé à réceptionner ces données rapidement afin de rendre la main à la sonde au plus tôt.

Le résultat d'une observation par une sonde est un ensemble de *mesures* (*samples*) correspondant aux différentes caractéristiques du sujet observé par la sonde. Une mesure est tout simplement un triplet formé d'un nom, d'une valeur et d'une estampille (*timestamp*) indiquant l'instant auquel la valeur a été mesurée. La classe `Sample` représente une telle mesure (cf. figure 5.10). Pour des raisons de performances, l'ensemble des mesures obtenues par une sonde lors d'une observation est regroupé dans un agrégat

```
public interface Sensor {
    String getName();
    void setListener(SamplesListener listener);
    void start();
    void stop();
    boolean isStarted();
}

public interface SamplesListener {
    void newSamples(Sensor source, SampleSet samples);
}
```

FIG. 5.9 – Les interfaces `Sensor` et `SamplesListener` du paquetage `org.obasco.wildcat.sensing`.

`SampleSet`. Chaque mesure est identifiée par un nom, unique dans un `SampleSet`. Au cours du temps, un même nom doit toujours correspondre – pour une sonde donnée – à la même caractéristique observée, mais l'ensemble des triplets renvoyés lors d'une mesure n'est pas forcément toujours le même.

```
public class Sample {
    public Sample(String name, Value value, long timeStamp) { ... }
    public String getName() { ... }
    public Value getValue() { ... }
    public long getTimeStamp() { ... }
}

public class SampleSet {
    public SampleSet() { ... }
    public SampleSet(long timeStamp) { ... }
    public long getTimeStamp() { ... }

    public void addSample(String name, Number value) { ... }
    public void addSample(String name, String value) { ... }
    public void addSample(String name, Boolean value) { ... }

    public String[] getSamplesNames() { ... }
    public Sample getSample(String name) { ... }
    public boolean hasSample(String name) { ... }
    public int size() { ... }
    public Sample[] getSamples() { ... }
}
```

FIG. 5.10 – Les classes `Sample` et `SampleSet` du pacquetage `org.obasco.wildcat.sensing`.

Créer une nouvelle sonde active se fait très simplement en créant une classe qui implémente l'interface `Sensor`. Lorsque la sonde est démarrée par un appel à sa méthode `start()`, la classe active un fil d'exécution asynchrone (`Thread`) chargé d'effectuer les mesures aux moments appropriés et d'en notifier le `SamplesListener`. Ce thread (ou tout du moins l'envoie de ces mesures) doit être stoppé lorsque la méthode `stop()` est invoquée.

La création d'un `SampleSet` se fait très simplement avec l'un des deux constructeurs ; celui ne prenant pas de paramètre crée une estampille correspondant au moment de sa création. La sonde peut ensuite facilement ajouter de nouvelles mesures au `SampleSet` avec l'une des nombreuses méthodes `addSample(...)` :

il en existe une pour chaque type primitif de Java, en plus de `Number`, `String` et `Boolean`.

### Sondes passives

Les sondes passives ont un comportement beaucoup plus simple que les sondes actives, et sont plus faciles à développer, puisqu'elles n'ont pas de fil d'exécution associé. Elles ne contiennent que le code nécessaire à l'observation du contexte. Concrètement, une sonde passive est un objet Java implémentant l'interface `Sampler` :

```
public interface Sampler {
    SampleSet sample();
}
```

Cette interface ne définit qu'une seule méthode, `sample()`, qui renvoie un `SampleSet`. À chaque invocation de `sample()`, la sonde effectue une nouvelle observation, et renvoie dans le `SampleSet` les valeurs courantes de différents caractéristiques qu'elle observe.

Afin d'unifier le traitement des différents types de sondes, WildCAT permet d'associer à une sonde passive une politique d'ordonnancement (voir la section 5.5.2 décrivant la configuration des sondes pour les différents types d'ordonnancement supportés). L'objet résultant de cette association (`SamplingSensor`) peut ensuite être considéré comme une sonde active et géré exactement de la même manière que les autres. Un démon est chargé d'invoquer les sondes ainsi « activées » en accord avec leurs politiques d'ordonnancement, en utilisant un ou plusieurs fils d'exécution.

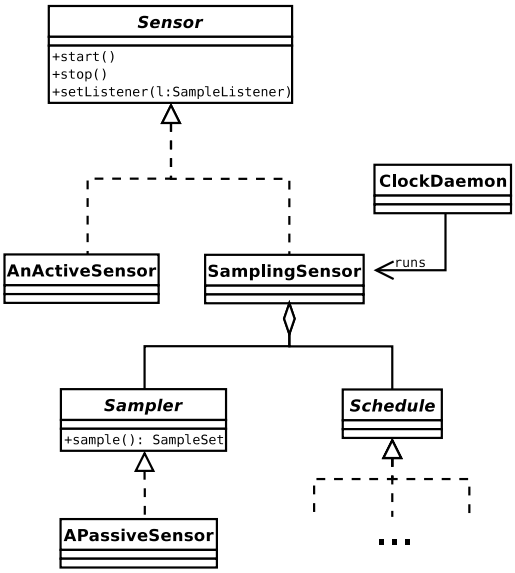


FIG. 5.11 – Les différents types de sondes.

### Protocole d'instanciation et de configuration

Quel que soit le type d'une sonde – active ou passive – le sujet spécifique qu'elle observe dépend non seulement de sa classe, mais aussi d'éventuels paramètres de configuration. Par exemple, on peut avoir une sonde `FileSystem` qui observe les caractéristiques d'un système de fichier (type, espace total, espace libre, taux de fragmentation...), et en créer plusieurs instances pour observer les différentes partitions.

Puisque les deux types de sondes sont définis par des interfaces Java, qui ne peuvent pas contenir de prototypes pour des constructeurs, le protocole d'instanciation des sondes est géré par une classe

séparée, `SensorFactory`. Celle-ci implémente le patron de conception *Fabrique* [Gamma et al., 1994]. Afin de garantir qu'une sonde ne peut pas être configurée partiellement, la configuration d'une sonde se fait uniquement lors de son instanciation, les informations de configuration étant passées au constructeur.

Suivant qu'une sonde est active ou non, et configurable ou non, quatre cas peuvent se produire :

1. Une sonde active non configurable doit définir un constructeur unique prenant une chaîne de caractère (`String`) en paramètre, correspondant à son nom : `<init>(String name)`.
2. Une sonde passive non configurable doit définir un constructeur unique ne prenant aucun paramètre.
3. Une sonde active configurable doit définir un constructeur prenant en paramètre une chaîne de caractère (`String`) et un `org.jdom.Element` représentant un fragment XML contenant ses paramètres de configuration. Elle peut aussi définir un constructeur ne prenant que la chaîne de caractère et utilisant des paramètres de configuration par défaut.
4. Une sonde passive configurable doit définir un constructeur prenant en paramètre un `Element` JDOM représentant un fragment XML contenant ses paramètres de configuration. Elle peut aussi définir un constructeur ne prenant que la chaîne de caractère et utilisant des paramètres de configuration par défaut.

Afin d'instancier et de configurer ces sondes, la classe `SensorFactory` définit les méthodes suivantes :

- `Sensor createSensor(String name, Class klass, Element configuration)` : crée une sonde active en instanciant la classe `klass` – qui doit implémenter l'interface `Sensor` – en passant `name` et `configuration` à son constructeur.
- `Sensor createSensor(String name, Class klass)` : crée une sonde active à partir de la classe `klass` – qui doit implémenter l'interface `Sensor` – en passant `name` à son constructeur. Si `klass` ne définit que le constructeur prenant en paramètre un nom et un élément XML, celui-ci est utilisé, le second paramètre étant `null`.
- `Sensor createSensor(String name, Class klass, Schedule sched, Element config)` : crée une sonde active à partir d'une sonde passive – anonyme – définie dans la classe `klass` qui doit implémenter `Sampler`. Une fois créée et configurée la sonde passive est activée en l'encapsulant dans un `SamplingSensor` qui y associe le nom `name` et la politique d'ordonnancement `sched`. L'instanciation de la classe `klass` se fait en utilisant son constructeur prenant un `Element` en paramètre.
- `Sensor createSensor(String name, Class klass, Schedule sched)` : crée une sonde active à partir d'une sonde passive – anonyme – définie dans la classe `klass` de la même façon que dans le cas précédent, mais en utilisant le constructeur sans paramètre de `klass`, ou, s'il n'existe pas, son constructeur prenant un `Element`, avec `null` pour valeur.
- `Sensor createSensor(Element descriptor)` : crée une sonde à partir d'un élément XML `sensor`, qui spécifie le nom, la classe, la politique d'ordonnancement et les informations de configuration. Cette méthode est chargée d'invoquer le constructeur approprié parmi les quatre précédents, en fonction des informations contenues dans le fragment XML. C'est le point d'entrée principal utilisé par le système lorsque l'on utilise les fichiers de configuration XML présentés dans la section 5.5.

### Sonde XML générique

L'implémentation de nouvelles sondes se fait normalement en définissant une nouvelle classe Java. Cependant, Java n'est pas toujours très bien adapté pour ce type de code. Certaines nécessitent de dialoguer directement avec des couches basses du système d'exploitation, voire avec le matériel, ce que Java ne permet pas, sauf par l'intérmédiaire de JNI, complexe à mettre en œuvre. D'autres ont besoin d'être extrêmement performantes pour ne pas perturber le fonctionnement du système, ou bien de communiquer avec des systèmes existant, accessibles souvent uniquement à partir de programmes C. Enfin, sur certains systèmes comme Linux, beaucoup d'informations système intéressantes sont disponibles sous forme de fichiers texte dans le système de fichier virtuel `/proc`, mais Java n'est pas conçu pour manipuler simplement du texte et nécessite souvent beaucoup de code pour effectuer des opérations simples[15].

[15] Cela est de moins en moins vrai à la vue des évolutions récentes de Java – avec par exemple l'apparition des expressions régulières dans la version 1.4 et d'une fonctionnalité similaire à `scanf()` dans la version 1.5. Cependant, Java est encore

Afin de faciliter la création de sondes dans d'autres langages plus appropriés, WildCAT fournit une sonde passive générique qui est capable d'invoquer n'importe quel programme externe et de récupérer ses mesures sous la forme d'un document XML. Cette sonde, définie dans la classe `XMLCommandSensor` et s'utilise de la façon suivante :

```
<sensor name="uptime" class="org.obasco.wildcat.sensors.XMLCommandSensor">
  <schedule>
    <periodic period="2000"/>
  </schedule>
  <configuration>
    <command>ruby</command>
    <parameter>/usr/local/share/wildcat/sensors/uptime.rb</parameter>
  </configuration>
</sensor>
```

Ainsi configurée, à chaque fois que le système voudra effectuer une nouvelle mesure (ici toutes les deux secondes), la sonde invoquera l'interprète Ruby avec le paramètre `.../sensors/memory.rb`. Ce programme (ici un script Ruby) doit envoyer sur sa sortie standard un document XML valide représentant le résultat de sa mesure. Ce document doit avoir la forme suivante :

```
<?xml version='1.0' encoding='UTF-8'?>
<sample-set timestamp='2004-09-29 12:04:42.342'>
  <sample name='uptime' type='number'>1051445.94</sample>
  <sample name='idle_total' type='number'>1012427.71</sample>
</sample-set>
```

Les types de données possibles pour les `sample`s sont `number` pour les nombres flottants, `string` pour les chaînes de caractères (sans guillemets) et `boolean` pour les booléens (`true` / `yes` / `false` / `no`).

Voici le script Ruby implémentant cette sonde :

```
#!/usr/bin/env ruby
require 'wildcat'

class UptimeSensor < Sampler
  def sample
    data = File.read('/proc/uptime').scanf("%f %f")
    return {'uptime' => data[0], 'idle_total' => data[1]}
  end
end

puts UptimeSensor.new.sense
```

Comme on peut le constater, dans ce cas particulier un langage comme Ruby est beaucoup mieux adapté que Java pour l'implémentation rapide d'une telle sonde. Bien entendu, la commande invoquée par la sonde XML générique peut être n'importe quel programme, écrit dans n'importe quel langage, en fonction de la tâche spécifique à réaliser.

### 5.6.2 Création d'une nouvelle implémentation

La création de nouvelles sondes utilisables avec l'implémentation standard fournie par WildCAT permet d'adapter facilement le framework au contexte spécifique d'une application. Cependant, cette solution n'est pas appropriée à certains domaines contextuels. En effet, l'implémentation standard de la notion de

loin de rivaliser avec des langages comme Perl, Ruby ou Python lorsqu'il s'agit de manipuler du texte.

domaine contextuel implique que toutes les ressources et attributs existent à tout moment en mémoire sous forme d'objets Java. Cette solution peut avoir un coût prohibitif dans certains cas où le domaine contextuel comprend des dizaines de milliers de ressources et d'attributs, dont seulement certains sont vraiment utiles pour l'application à un moment donné. De plus, il existe déjà de nombreux systèmes qui implémentent l'essentiel des fonctionnalités d'un domaine contextuel et qu'il pourrait être utile d'intégrer à WildCAT sans avoir à dupliquer ces fonctionnalités, comme par exemple FAM pour le monitoring de fichier, ou SNMP pour l'observation des réseaux.

Pour tous ces cas de figure, la meilleure solution consiste à créer une nouvelle implémentation *ad hoc* de domaine contextuel. WildCAT a été conçu explicitement pour permettre ce genre d'approche ; d'ailleurs l'implémentation fournie en standard n'est qu'un exemple de cette approche, et n'est pas traitée de façon particulière par le système.

Pour créer une nouvelle implémentation de domaine contextuel, il « suffit » d'implémenter l'interface `ContextDomain` :

```
public interface ContextDomain {
    void initialize(Context ctx);
    String getName();
    boolean exists(Path path);
    Value resolve(Path attr);
    Path[] getChildren(Path res);
    Path[] getAttributes(Path res);
    void register(ContextListener listener, int eventKinds, Path path);
    void unregister(ContextListener listener, int eventKinds, Path path);
    void update(Path path, Path cause);
}
```

Une fois implémentée correctement, son intégration dans WildCAT se fait extrêmement simplement, exactement comme avec l'implémentation standard :

```
Context ctx = new Context();
ContextDomain domain = new MonDomaineContextuel("nom", config, ...);
ctx.addDomain(domain);
```

Cette interface est entièrement définie par rapport à la notion abstraite de chemin, représentée par la classe `Path`. Les utilisateurs d'un domaine contextuel (y compris la classe centrale de WildCAT, `Context`) n'ont pas accès à la représentation interne des notions de ressources et d'attributs spécifiques à un domaine.

Puisque `ContextDomain` est une interface et non une classe, son initialisation ne peut pas être spécifiée par un constructeur. Elle se fait par l'intermédiaire de la méthode `initialize(Context)`, invoquée par le contexte lorsque le domaine contextuel y est ajouté par `addDomain()`.

Les cinq méthodes suivantes (`getName()`, `exists()`, `resolve()`, `getChildren()`, `getAttributes()`) correspondent à l'implémentation du modèle logique de WildCAT (cf. figure 5.3, page 78) et leur signification est simple à comprendre. Les deux suivantes (`register()` et `unregister()`) doivent implémenter le mécanisme de notification des clients lorsque certains événements se produisent à l'intérieur du domaine.

La dernière méthode, `update(Path path, Path cause)`, est utilisée pour notifier un domaine contextuel qu'un de ses éléments, désigné par `path` doit être mis à jour suite à une modification d'un autre élément `cause`. Cette méthode fait partie d'un système permettant à différents domaines contextuels de faire dépendre leurs données les uns des autres sans avoir à connaître leurs implémentations respectives :

1. Lors de son initialisation, dans la méthode `initialize()`, chaque domaine contextuel peut récupérer une référence vers le gestionnaire de dépendance inter-domaines. Il suffit pour cela d'invoquer la méthode `getDependencyManager()` du `Context` passé en paramètre, qui renvoie un `DependencyManager` (cf. figure 5.12).
2. À chaque fois qu'un événement se produit dans le domaine contextuel, celui-ci doit en informer le gestionnaire de dépendances à travers son interface `ContextListener`.

3. Si un domaine contextuel contient des données (par exemple des attributs) qui dépendent d'autres domaines, il doit en informer le gestionnaire de dépendance en utilisant sa méthode `registerDependency(src, dest)`, où `src` est le chemin (externe) dont l'élément local `dest` dépend. Cette dépendance peut être supprimée par un appel à `unregisterDependency(src, dest)`.
4. Lorsqu'une telle dépendance a été enregistrée, le domaine contextuel sera prévenu par le gestionnaire de dépendance – à chaque fois qu'un événement concernant `src` lui sera notifié – par un appel à sa méthode `update()`, où le premier paramètre (`path`) correspond à `dest` et le second (`cause`) à `src`.

```
public class DependencyManager implements ContextListener, Runnable {
    void registerDependency(Path src, Path dest) { ... }
    void unregisterDependency(Path src, Path dest) { ... }
}
```

FIG. 5.12 – La classe `DependencyManager`, responsable des dépendances inter-domaines contextuels.

Ainsi, WildCAT est capable de faire non seulement *cohabiter* mais aussi *coopérer* un nombre quelconque d'implémentations différentes de la notion abstraite de domaine contextuel, chacune étant parfaitement adaptée à sa tâche et à ses contraintes spécifiques.

## 5.7 Conclusion

Après avoir identifié le besoin des applications adaptatives en terme d'observation du contexte, nous avons dégagé les objectifs d'un tel service et des critères qui permettent d'en évaluer la qualité.

Nous présentons ensuite WildCAT, en décrivant tout d'abord le modèle utilisé pour représenter les information contextuelles, puis les deux interfaces de programmation permettant aux programmeurs d'application d'accéder à ce modèle. La première de ces interfaces permet d'interroger ponctuellement le service pour découvrir la structure du contexte et les valeurs des éléments qui le composent. La seconde permet à une application de s'enregistrer auprès du service afin d'être ensuite notifiée de façon asynchrone à chaque fois que certaines conditions, définies par l'application, se produisent.

Notre solution se présente sous la forme d'une cadre de programmation (*framework*) qui doit être configuré spécifiquement pour chaque application (bien que certains éléments de cette configuration puissent facilement être réutilisés par plusieurs applications). Nous décrivons donc ensuite les interfaces de configuration et d'extension du service, qui se présentent respectivement sous la forme de fichiers XML et d'interfaces Java qui doivent être implémentées pour augmenter les fonctionnalités du système. La figure 5.13 représente l'intégralité du framework WildCAT.

Le résultat obtenu est un système générique facilement extensible qui offre une interface de programmation relativement simple à utiliser et qui permet donc de rendre une application Java très facilement sensible à son contexte d'exécution, condition *sine qua non* pour être adaptive.

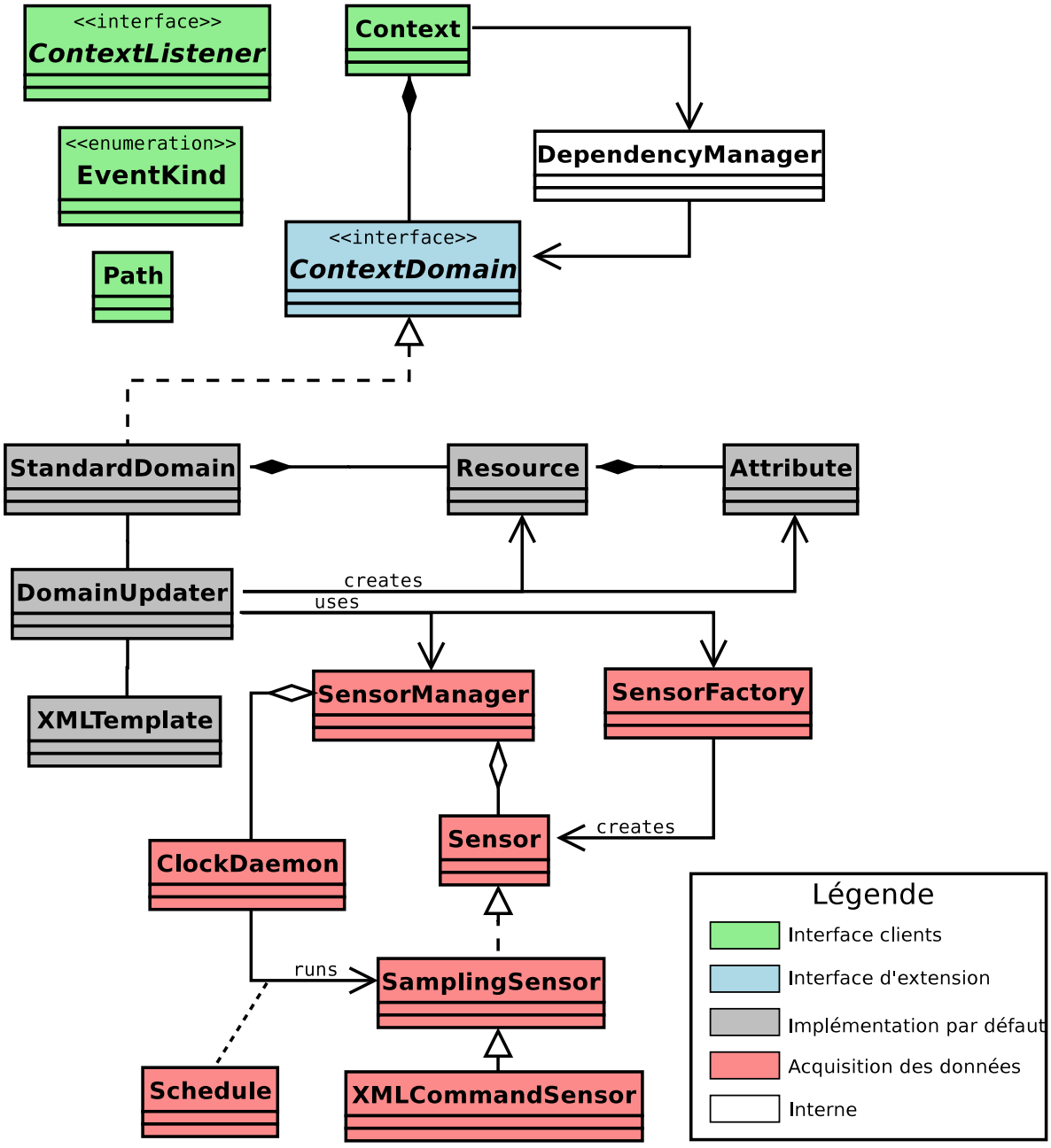


FIG. 5.13 – Diagramme UML complet de WildCAT .

# Chapitre 6

# FScript : un langage dédié pour la reconfiguration consistante de composants Fractal

*I cannot say whether things will get better if we change ;*
*what I can say is they must change if they are to get better.*

— G. C. Lichtenberg

*I can't change the direction of the wind,*
*but I can adjust my sails to always reach my destination.*

— James Dean

## Sommaire



Ce chapitre décrit un langage dédié à la spécification de reconfigurations dynamiques structurelles de composants et d'applications Fractal. Ce langage est un élément essentiel de notre solution pour la création d'applications adaptatives, puisque ce sont ces reconfigurations, appliquées aux moments appropriés, qui vont adapter l'application à ses conditions d'exécution variées et changeantes.

## 6.1 Introduction

L'adaptation d'une application passe obligatoirement par une « reconfiguration », qui peut être de différentes natures comme le montrent les exemples décrits dans l'état de l'art au chapitre 3. Nous avons déjà présenté et justifié lors de la conclusion de ce chapitre notre choix de considérer essentiellement des adaptations *structurelles*, c'est-à-dire qui reconfigurent l'architecture de l'application, modifiant ainsi indirectement son comportement et la qualité de son fonctionnement. Le modèle de composant Fractal supporte plusieurs mécanismes pour structurer les applications : connexions entre interfaces, composition hiérarchique généralisée, et lien méta (grâce à notre extension). L'un des principaux intérêts de Fractal qui nous a conduit à le choisir comme « substrat » pour l'adaptation est que tous ces mécanismes sont entièrement réflexifs et dynamiques, et peuvent donc être utilisés pour reconfigurer – et donc adapter – les applications en cours d'exécution.

Bien qu'il supporte toutes les fonctionnalités dont nous avons besoin, le modèle Fractal est spécifié sous la forme d'un ensemble d'interfaces de programmation (APIs). Cela le rend particulièrement adapté à la création d'outils (de développement, de déploiement, de gestion...), mais cette forme ne convient pas à notre mode d'utilisation :

1. Utiliser directement ces interfaces pour programmer les reconfigurations, par exemple en Java, élimine toute possibilité d'offrir des **garanties** quant à l'effet de ces reconfigurations sur l'application. Par exemple, une propriété aussi fondamentale que la *terminaison* d'une reconfiguration ne peut tout simplement pas être garantie si les reconfigurations sont spécifiées dans un langage généraliste. Comme le montre le tableau de synthèse 3.1, les seules approches qui permettent d'offrir de bonnes garanties sont celles qui sont basées sur des modèles de composants *ad hoc* aux opérations de reconfigurations restreintes comme par exemple ACEEL (Section 3.3.4, page 39).
2. Les interfaces de programmation de Fractal sont conçues pour être minimales et orthogonales, et leur utilisation en pratique est souvent très verbeuse. Cela est d'autant plus vrai dans le cas d'un langage typé statiquement et explicitement comme Java, qui nécessite de nombreux transtypages (*casts*) qui rendent le code rapidement illisible. La gestion explicite et obligatoire des exceptions est aussi un élément qui alourdit le code. Il en résulte que même des opérations conceptuellement simples nécessitent un nombre de code disproportionné. Par exemple pour ajouter le composant `child` au composite `parent`, le code correspondant devrait en tout rigueur être le suivant (la gestion d'erreur n'étant pas spécifiée) :

```
LifeCycleController lc = null;
try {
  lc = (LifeCycleController) parent.getFcInterface("lifecycle-controller");
} catch (NoSuchInterfaceException nsie) { /* Do nothing. */ }
boolean started = (lc != null) ? "STARTED".equals(lc.getFcState()) : false;
if (started) {
  try { lc.stopFc(); }
  catch (IllegalLifeCycleException ilce) { /* Handle error. */ }
}
try {
  ContentController cc = (ContentController) parent.getFcInterface("content-controller");
  try { cc.addFcSubComponent(child); }
  catch (IllegalLifeCycleException ilce) { /* Should not happen. */ }
  catch (IllegalContentException ice)    { /* Handle error. */ }
} catch (NoSuchInterfaceException nsie)  { /* Handle error. */ }
finally {
  if (started) {
    try { lc.startFc(); }
    catch (IllegalLifeCycleException ilce) { /* Handle error. */ }
  }
}
```

3. Enfin, le modèle Fractal introduit des notions – composants et interfaces – qui n'existent pas directement dans le langage hôte (Java dans notre cas) mais n'étend pas la syntaxe de celui-ci pour faciliter

leur manipulation. Or, ces notions étendent le « langage » dans lequel les programmes sont écrits ; on peut même considérer Fractal comme une extension du modèle objet de Java. Cependant, sans extension syntaxique comme celles présentes par exemple dans ArchJava [Aldrich et al., 2002], les composants et interfaces doivent être manipulés comme des objets Java normaux. Il en résulte une certaine confusion pour le programmeur entre les différents niveaux de discours qui se mélangent. Le meilleur exemple de cette situation est sans doute la présence de deux notions d'interfaces : celle de Java (ensemble de signatures de méthodes) et celle de Fractal (service offert ou requis par un composant). Ces deux notions sont différentes, mais portent le même nom. Le fait que les interfaces Fractal soient représentées par des objets qui implémentent certaine(s) interface(s) Java ne fait qu'ajouter à la confusion.

Ces trois éléments – manque de garanties, notation verbeuse et abstractions non représentées directement – correspondent précisément aux problèmes que permet de résoudre la création d'un langage dédié :

1. Un langage dédié peut avoir un pouvoir d'expression volontairement limité pour permettre d'offrir les garanties nécessaires.
2. Puisque qu'un tel langage se limite à un domaine particulier, il peut utiliser des notations spécifiques qui le rendent moins verbeux et plus lisible.
3. Enfin, il peut intégrer directement au niveau du langage les notions spécifiques à son domaine, et uniquement celles-ci.

C'est pourquoi nous avons défini FScript, *un langage dédié à la spécification et à l'exécution de reconfigurations structurelles d'applications et de composants Fractal*. Comme son nom l'indique, FScript peut être considéré comme un « langage de script » qui permet de programmer des scripts de reconfiguration, mais dont le *pouvoir d'expression*, la *syntaxe* et *l'implémentation* ont été conçus pour offrir toutes les propriétés dont nous venons de parler.

La section 6.2 constitue une étude de domaine dans laquelle nous identifions les fonctionnalités que doit exhiber un tel langage, les concepts spécifiques à ce domaine et les différentes garanties que le langage doit offrir, sous la forme de critères de consistance des reconfigurations. Nous présentons ensuite FPath (Section 6.3), un sous-ensemble du langage FScript conçu pour *naviguer* dans les architectures Fractal mais qui ne permet pas de *modifier* ces architectures. La section 6.4 décrit le reste du langage FScript, et en particulier la possibilité de définir des *actions de reconfiguration*. Enfin, dans la section 6.5 nous décrivons l'implémentation de FScript et le modèle d'exécution qui permet de garantir la consistance des reconfigurations, avant de conclure le chapitre (Section 6.6). Afin de ne pas alourdir ce chapitre, certains détails plus techniques ne sont décrits que dans l'annexe A.

## 6.2 Étude de domaine

Dans cette section, nous étudions les spécificités du domaine visé par notre langage dédié FScript, à savoir la manipulation de l'architecture d'applications constituées de composants Fractal.

### 6.2.1 Fonctionnalités requises & pouvoir d'expression

L'objectif du langage est de permettre d'exprimer de façon naturelle des transformations sur la structure et l'état (configuration) d'un ensemble de composants Fractal, tout en offrant des garanties quant à la consistance de ces manipulations. Notre langage doit permettre d'accéder à l'ensemble de l'interface de programmation Fractal, y compris nos propres extensions. On peut distinguer parmi les services offerts par ces interfaces les domaines suivants :

– navigation et introspection de la structure des composants : découverte des interfaces (`component`), du contenu des composites (`content-controller` et `super-controller`), navigation le long des connexions entre interfaces (`binding-controller`) et du lien méta (`metalink-controller`).

- introspection d'une partie de l'état des composants : cycle de vie (`lifecycle-controller`), paramètres de configuration (`attribute-controller`) et nommage (`name-controller`)
- modification de la structure : modification du contenu (`content-controller`), des connexions (`binding-controller`), du lien méta (`metalink-controller`) et création de nouveaux composants (`factory`)[1].
- modification de l'état : changements des paramètres de configuration (`AttributeController`), manipulation du cycle de vie (`LifeCycleController`). Notons que Fractal ne permet d'accéder qu'à une partie limitée de ce qui constitue l'*état* d'un composant : l'état des objets Java correspondant à l'implémentation des composants primitifs n'est pas accessible. Cette limitation implique qu'il n'est pas possible avec le méthodes fournies par Fractal de transférer l'état d'un composant primitif à un autre, opération nécessaire pour pouvoir remplacer des composants à l'exécution. Rien n'empêche d'étendre Fractal pour y ajouter cette fonctionnalité (on peut imaginer une nouvelle interface de contrôle `serialization-controller` par exemple), mais puisqu'aucune implémentation n'existe actuellement à notre connaissance, nous ne traiterons pas explicitement de cette problématique. Si une telle extension voit le jour dans le futur, son intégration pourrait se faire très simplement dans notre langage de reconfiguration sous la forme d'une ou plusieurs nouvelles primitives de reconfiguration.

Une partie des méthodes définies dans l'interface de programmation Fractal est purement sans effet de bord et servent uniquement à la découverte et à la navigation (introspection). Les autres méthodes servent quant à elles à modifier les composants. Une interface de contrôle typique mélangent les deux types de méthodes. En pratique, les méthodes d'introspection sont celles qui renvoient une valeur, alors que toutes celles qui modifient les composants ont un type de retour `void`. Les deux types de méthodes signalent d'éventuelles erreurs en levant des exceptions.

On peut voir les méthodes des différentes interfaces proposées par Fractal comme un ensemble d'*opérations primitives*, certaines étant purement fonctionnelles (introspection) et les autres n'existant que pour leurs effets de bord sur les composants (intercession). Cette distinction entre les deux types d'opérations est important pour pouvoir offrir des garanties sur l'effet des scripts sur une application Fractal. Les différentes interfaces de contrôle de Fractal sont conçues pour être minimales et orthogonales entre elles. Si nous voulons que les utilisateurs de notre langage puissent avoir un pouvoir d'expression suffisant, ce dernier doit permettre d'invoquer la totalité de ces opérations et de les combiner.

Il doit être possible d'exprimer des reconfigurations génériques, dans lesquelles certains éléments sont *variables* et découverts dynamiquement. Ainsi, un script qui déconnecte l'ensemble des interfaces serveur d'un composant donné ne doit pas dépendre du type particulier de ce composant (nombre et types de ces interfaces). Si cela était le cas, un script spécifique devrait être écrit pour chaque type de composant différent présent dans l'application. Notre langage doit donc permettre d'*appliquer une opération à un ensemble d'éléments* (composants, interfaces...) découverts dynamiquement grâce aux fonctions d'introspection.

La possibilité d'écrire des reconfigurations génériques est importante en tant que telle, mais devient beaucoup plus intéressante lorsqu'il est possible de *réutiliser* ce code, c'est-à-dire des (parties de) scripts de reconfigurations entre applications. Il devient alors possible de créer des bibliothèques de scripts plus ou moins génériques et réutilisables dans plusieurs applications. Étant donné notre objectif final qui consiste à utiliser ces scripts de reconfiguration dans des stratégies d'adaptation, cette fonctionnalité est particulièrement importante si l'on veut pouvoir écrire facilement des stratégies générales (par exemple une stratégie de répartition de charge).

[1] Tout comme Java, Fractal ne définit pas d'opération explicite de destruction de composants. La seule façon de retirer un composant d'une application est de supprimer toutes ses relations (connexions, composition, lien méta...) de tous les composants de l'application et de l'arrêter. Rien dans Fractal ne garantit qu'il sera alors détruit, seulement qu'il n'aura plus d'impact sur l'application.

### 6.2.2 Concepts spécifiques à la manipulation de composants Fractal

Le modèle de composants Fractal introduit un ensemble de concepts qui lui sont spécifiques et qui n'existent habituellement pas dans les langages de programmation objet. Notre langage de script doit nous permettre de manipuler de façon la plus naturelle possible ces concepts, ou au moins un partie d'entre eux.

Une première analyse de la spécification du modèle Fractal permet d'identifier les concepts suivants :
- interfaces de composants ;
- types d'interface ;
- types de composants.

Étrangement, la notion de composant elle-même n'apparaît pas explicitement au niveau du modèle – au sens où elle n'est pas réifiée – puisqu'un composant y est désigné que par une interface particulière (`Component`). La notion de connexion n'est pas non plus réifiée (contrairement à certains autres modèles de composants [Medvidovic and Taylor, 2000]), mais toutes les informations nécessaires sont disponibles à travers l'interface `BindingController`.

Bien qu'elles soient pour leur part explicitées, nous considérons que les deux notions de types supportées par Fractal (`InterfaceType` et `ComponentType`) ne sont pas véritablement des concepts du modèle à part entière :
- Le type d'un composant est seulement l'ensemble de ses interfaces externes, et n'apporte donc aucune information supplémentaire dans le modèle.
- Le type d'une interface ajoute effectivement des informations qui décrivent les spécificités d'une interface, mais il s'agit plus de « méta-données » la concernant que d'un concept séparé. En réalité, la seule raison pour séparer ces informations de l'interface elle-même est de permettre l'utilisation de systèmes de types différents tout en conservant la généralité du noyau du modèle. Dans notre cas, nous utiliserons le système de type standard ; il n'y a donc aucune raison de considérer le type d'une interface comme un concept à part entière.

En plus des aspects purement structurels, notre langage de reconfiguration doit pouvoir manipuler une partie de l'état des composants, à savoir leur cycle de vie et leurs paramètres de configuration. Le cycle de vie que nous utilisons se limite à deux états, *démarré* ou *stoppé*, et peut donc être représenté par un simple booléen. Le cas des paramètres de configuration est plus complexe car bien qu'ils ne soient pas réifiés dans le modèle Fractal, on peut les considérer comme des « objets » spécifiques constitués d'un nom et d'une valeur associés à un composant.

Pour conclure, les concepts spécifiques que nous choisissons comme devant être représentés explicitement dans notre langage sont donc :

1. Les *composants*, identifiés en tant que tels (correspond en pratique à l'interface `Component`).
2. Les *interfaces* de composants, y compris toutes les méta-données les décrivant sous la forme des `InterfaceType`s.
3. Les *paramètres de configuration*, qui correspondent à une paire de méthodes (parfois une seule dans le cas des attributs en lecture seule) d'une interface `AttributeController`.

### 6.2.3 Critères de consistance

Nous avons déjà indiqué plus haut que l'un des objectifs prioritaires de la conception et de l'implémentation du langage FScript est de garantir la consistance des reconfigurations appliquées. Dans cette section, nous détaillons et justifions les différents critères de consistance retenus.

La consistance des reconfigurations dynamiques a été relativement peu étudiée comparativement aux développement de techniques pour *permettre* ces reconfigurations. On retrouve encore une fois la dialectique entre forme (garanties) et ouverture (mécanismes pour la reconfiguration dynamique), mais pour l'instant les recherches se sont attachées à rendre le logiciel de plus en plus flexible. Quelques travaux ont cependant essayé de rétablir l'équilibre en recherchant les critères de consistance nécessaires et suffisants pour pouvoir effectuer des reconfigurations dynamiques de façon sûre. [Wermelinger and Fiadeiro, 2002]

utilise une approche formelle à base de transformation de graphes (représentant l'architecture de l'application) pour spécifier les reconfigurations et vérifier leurs propriétés. [Cook and Dage, 1999] étudie le cas particulier de la mise-à-jour sûre de composants, par exemple pour installer une nouvelle version ou corriger un bogue. [Moazami-Goudarzi, 1999] s'intéresse aux critères de consistance qui permettent de garantir la conservation de l'intégrité de l'application face à des reconfigurations dynamiques, alors qu'[Almeida, 2001] étudie les caractéristiques des algorithmes de reconfiguration (« *change management* ») qui permettent de garantir ces critères. Enfin, [Tarr and Clarke, 1997] s'intéresse au processus de gestion de la consistance à un niveau plus global : où, comment et par qui sont définies le contraintes, comment sont gérées les erreurs, etc.

Il se dégage de ces travaux deux types de critères : d'une part ceux qui s'appliquent au *processus* de reconfiguration [Almeida, 2001], et d'autre part ceux qui s'appliquent aux *configurations* initiales et finales de l'application [Moazami-Goudarzi, 1999]. La consistance d'une reconfiguration inclut, mais n'est pas limité à, la consistance des états initiaux et finaux (dans notre cas, ce nouvel état est censé être mieux *adapté* aux contexte d'exécution). Le processus de reconfiguration peut être vu comme une *transaction*. Nous nous sommes donc en partie inspirés des critères utilisés dans le monde des bases de données pour définir l'ACIDité[2] des transactions. Nous retenons pour le processus les critères d'*atomicité*, d'*isolation* et de *consistance des états*, auquel nous ajoutons la *terminaison* (la durabilité n'a pas de sens dans notre cas). Reste à préciser le sens du critère de consistance des états dans le cas particulier des configurations. Certains critères de consistance sont déjà définis au niveau du modèle Fractal général (compatibilité entre interfaces par exemple). Cependant, une configuration Fractal valide du point de vue du modèle n'est pas forcément valide du point de vue fonctionnel. Nous considérons avec [Tarr and Clarke, 1997] qu'un système de reconfiguration dynamique sûr doit être capable de prendre en compte des critères spécifiques à l'application reconfigurée.

Plus précisément, les critères que nous retenons sont les suivants :

**Terminaison.** Le premier critère, qui est aussi le plus fondamental, est la *terminaison* des reconfigurations. Puisque les reconfigurations s'appliquent pendant l'exécution de l'application, et que certaines peuvent impliquer de stopper des composants, des reconfigurations qui ne terminent pas, par exemple pour cause d'interblocage (*deadlock*) ou de cycle infini, risquent de rendre l'intégralité de l'application inutilisable.

**Atomicité.** Une reconfiguration doit être considérée comme un tout cohérent, une suite d'opérations qui font passer l'application d'un état initial à un état final différent. Il est important que l'application reste toujours dans un état cohérent afin de pouvoir fonctionner correctement. Or seul l'état initial est *a priori* garanti consistant, puisqu'il s'agit soit de l'état d'origine de l'application avant toute reconfiguration, soit du résultat d'une reconfiguration précédente qui était elle-même consistante. Si une reconfiguration ne résulte pas dans un nouvel état consistant, elle ne doit pas être appliquée du tout. Malheureusement, il n'est toujours possible de prévoir à l'avance la consistante du résultat, et des erreurs inattendues peuvent se produire pendant la reconfiguration. Dans ce cas, le système de reconfiguration ne doit pas laisser l'application dans un état intermédiaire potentiellement inconsistant[3] et doit revenir à l'état initial qui est le seul garanti consistant. Une reconfiguration doit donc être *atomique*, c'est-à-dire que soit elle est appliquée intégralement et résulte dans un nouvel état correct, soit elle n'est pas appliquée du tout et le système se retrouve dans l'état initial (tout ou rien).

**Isolation.** Si plusieurs reconfigurations se produisent en même temps dans une application donnée, les états intermédiaires potentiellement inconsistants d'une reconfiguration ne doivent pas être visibles des autres. L'état de l'application du point de vue de chaque reconfiguration ne doit refléter que les changements explicitement effectués par cette reconfiguration.

**Consistance des états.** Ce critère indique que l'état initial et l'état final d'une reconfiguration sont tous les deux consistants individuellement, et que la transition elle-même est consistante.

[2] **A**tomicité, **C**onsistance, **I**solation et **D**urabilité.

[3] Certains états intermédiaires peuvent être consistants, mais même si c'est le cas ils ne correspondent pas au résultat attendu par l'initiateur de la reconfiguration. Les conserver peut alors conduire à des problèmes ultérieurs ou d'autres reconfigurations ne sont plus applicables.

Dans notre cas, la consistance d'une configuration Fractal peut être vue à deux niveaux :

1. Au niveau du modèle lui-même, Fractal définit essentiellement une contrainte structurelle : pour qu'un composant puisse fonctionner (être dans l'état `STARTED`), toutes ses interfaces clientes obligatoires doivent être connectées à des interfaces compatibles.
2. Au niveau de la sémantique spécifique de l'application, notre extension décrite dans la section 4.3.2 permet d'ajouter dynamiquement de nouvelles contraintes structurelles spécifiques.

Quant à la consistance de la transition, il s'agit de garantir l'*intégrité comportementale* de l'application qui est reconfigurée en cours d'exécution : notre système de reconfiguration doit faire en sorte d'interférer le moins possible avec le comportement de l'application. En particulier, il est essentiel ne pas perdre de messages pendant la reconfiguration : si un composant qui participe à une reconfiguration reçoit un message pendant cette reconfiguration à un moment où il ne peut pas le traiter, le message doit être mis en attente et relancé lorsque le composant est de nouveau prêt.

Concernant les stratégies de vérification et d'application de ces critères, nous choisissons dans un premier temps de nous reposer sur des techniques de vérification dynamique, décrites dans la section 6.5. Cependant, nous avons conçu FScript et en particulier son pouvoir d'expression limité afin de permettre dans le futur d'utiliser des techniques d'analyses statiques de code afin de détecter les reconfigurations invalides au plus tôt.

## 6.3 Navigation dans les architectures Fractal avec FPath

Dans cette section, nous présentons une notation simple et expressive pour la navigation dans une architecture Fractal et la sélection d'éléments (composants, interfaces, attributs) répondant à certains critères, *sans modifier l'architecture*. Ce langage, nommé FPath, est inspiré du langage XPath [World Wide Web Consortium, 1999] utilisé pour remplir les mêmes besoins dans le cadre des documents XML. Il repose sur la modélisation d'un ensemble de composants Fractal sous la forme d'un graphe orienté, dont les nœuds représentent les composants, interfaces et attributs, et dont les arcs sont annotés par des labels qui dénotent le type de relation entre deux nœuds.

Bien que cette notation ait été développée dans le cadre de la définition du langage dédié à la reconfiguration de composants, dont elle constitue un sous-ensemble, les besoins auxquels elle répond sont plus généraux et une telle notation peut être utilisée dans des contextes différents. C'est pour cette raison que nous la décrivons de façon autonome dans cette section, et que son implémentation a été conçue de manière à pouvoir être utilisée facilement dans d'autres contextes. Par exemple, l'extension du modèle Fractal décrite dans la section 4.3.2 utilise cette notation pour exprimer les contraintes architecturales associées aux composants.

FPath ne permet d'écrire que des *expressions*, et ne dispose pas de structures de contrôle. Une expression FPath s'évalue sans effet de bord et retourne toujours une *valeur*, qui peut être d'un des types suivants :

– nombres, chaîne de caractère et booléens ;
– ensemble homogènes de composants, d'interfaces ou d'attributs.

Seuls les trois derniers types sont spécifiques à FPath, les trois premiers étant des types généraux « classiques ». De la même manière, FPath supporte en plus des constructions spécifiques au domaine des expressions habituelles que l'on trouve dans la plupart des langages : expressions arithmétiques et booléennes, comparaisons et invocation de fonctions.

### 6.3.1 Chemins et types de données associés

Un *chemin* est une expression spécifique à FPath. C'est ce type d'expression qui permet la navigation et la sélection d'éléments dans les architectures Fractal. Ces expressions de chemins sont inspirées expressions correspondantes du langage XPath [World Wide Web Consortium, 1999], mais utilisent un modèle sous-jacent différent : là où XPath utilise une représentation d'un document XML sous forme de graphe, FPath utilise le même genre de représentation, mais pour modéliser des ensembles de composants Fractal.

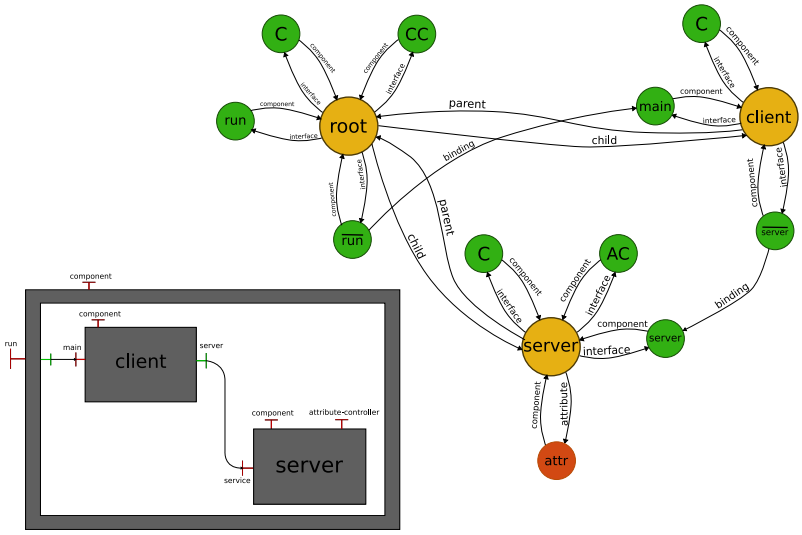


FIG. 6.1 – Exemple de modélisation d'un ensemble de composants Fractal sous forme de graphe.

**Structure du graphe.** Étant donné un ensemble de composants Fractal constituant une application, le graphe correspondant est constitué d'un ensemble de nœuds qui peuvent être de trois types différents correspondant aux trois concepts spécifiques au domaine Fractal : chaque composant Fractal est représenté par un unique nœud de type *composant*, chaque interface de composant – interne ou externe – est représentée par un nœud de type *interface*, et chaque paramètre de configuration de composant (correspondant à une ou deux méthodes d'une interface `attribute-controller`) est représenté par un nœud de type *attribut*. Ces nœuds sont reliés par un ensemble d'arcs orientés, chacun portant un *label* identifiant le type de relation entre les deux nœuds qu'il relie. Les différents labels possibles pour un arc sont :

| | |
|---|---|
| **component** | Relie un nœud *interface* ou *attribut* à celui du *composant* auquel appartient l'interface ou l'attribut. |
| **interface** | Relie un nœud *composant* à chacune de ses interfaces. |
| **attribute** | Relie un nœud *composant* à chacun de ses paramètres de configuration. |
| **binding** | Relie deux *interfaces* si et seulement si les deux interfaces sont connectées. L'orientation de l'arc correspond à l'orientation de la connexion. |
| **child** | Relie deux *composants* $A$ et $B$, dans cette direction, si et seulement si $B$ est un sous-composant direct du composite $A$. |
| **parent** | Symétrique de child. Relie les *composants* $A$ et $B$ si et seulement si $B$ est un des parents (super-composants) directs de $A$. |
| **meta** | Relie un *composant* de base à son méta-*composant*, s'il existe. |

Dans la suite, nous identifierons les composants, interfaces et attributs aux nœuds qui les représentent pour alléger le discours. La figure 6.1 illustre sur un exemple la modélisation correspondante à un ensemble de composants Fractal.

**Structure d'un chemin.** Étant donnée cette modélisation et un nœud de départ quelconque (un chemin est toujours relatif), un *chemin* FPath permet de sélectionner un ensemble de nœud qui répondent à certains critères en navigant le long des arcs. Pour cela, un chemin est constitué d'un ensemble de *pas* successifs, séparés par une barre oblique. Chacun de ces pas est lui-même structuré en trois éléments : un *axe*, un *test* et une suite, éventuellement vide, de *prédicats*. La syntaxe concrète d'un chemin est :

```
axe1::test1[pred1]/axe2::test2[pred2]/...
```

- L'*axe* est un identifiant parmi un ensemble fini qui correspond aux labels possibles pour les arcs du graphe.
- Le *test* est soit un nom (identifiant), soit une étoile `*`.
- Les *prédicats* sont des expressions FPath complètes évaluées pour leur valeur booléenne.

Par exemple, le chemin `sibling::*/interface::*[provided(.)][not(bound(.))]` est constitué de deux pas. Le premier a pour axe `sibling`, pour test `*` et n'a pas de prédicat. Le second a pour axe `interface`, pour test `*` et a deux prédicats.

**Évaluation d'un chemin.** Étant donné le nœud de départ, la valeur du chemin est déterminée en évaluant successivement chaque pas, et en utilisant comme nœuds de départ pour un pas donné l'ensemble des nœuds renvoyés par le pas précédent. Plus précisément, l'algorithme est le suivant :

**C1.** [Initialisation] On initialise le compteur $i \leftarrow 1$, et l'ensemble $current \leftarrow \{\text{nœud de départ}\}$.

**C2.** [Pas] On initialise $next \leftarrow \emptyset$, puis pour chaque nœud $c$ de $current$ :

**C2.1.** [Évaluation] On évalue le pas $step_i$ avec $c$ pour nœud courant, et on note $result$ l'ensemble de nœuds résultant.

**C2.2.** [Accumulation] $next \leftarrow next \cup result$

**C3.** [Itération] $current \leftarrow next$. Si $i < n$ (où $n$ est la taille du chemin), alors $i \leftarrow i + 1$ et on retourne à l'étape C2. Sinon, l'algorithme se termine et renvoie l'ensemble $current$.

Puisqu'il n'est possible de construire que des chemins de taille finie, l'algorithme termine toujours.

**Évaluation d'un pas.** L'évaluation d'un pas se fait en deux temps. La première étape, consiste, à partir des nœuds de départ, à suivre l'*axe* désigné afin de sélectionner un nouvel ensemble de nœuds. La seconde étape réduit cet ensemble de candidats en ne retenant que ceux dont le nom correspond au *test*, et pour lesquels tous les prédicats sont *vrais*. L'algorithme est donc le suivant ($c$ est le nœud courant) :

**P1.** [Initialisation] $result \leftarrow \emptyset$.

**P2.** [Sélection] On sélectionne dans un premier temps tous les nœuds du graphe qui sont connectés au nœud courant $c$ par un arc dont le label est $axe$ : $result \leftarrow \cup\{n : c \xrightarrow{axe} n\}$.

**P3.** [Test] Si le test est un identifiant (par opposition à `*`), on retire de $result$ les nœuds dont le nom est différent de cet identifiant : $result \leftarrow \{n \in result : name(n) = test\}$. $name()$ renvoie le nom d'un nœud : `getFcItfName()` pour les interfaces, le nom du paramètres (`foo` s'il est accessible par la méthode `getFoo()`) pour les nœuds attributs, et pour les composants $name()$ renvoie leur nom tel qu'indiqué par leurs `name-controller`s, ou une chaîne vide s'ils n'en ont pas.

**P4.** [Filtrage] On applique à chaque élément de $result$ la conjonction des prédicats, en on ne retient que les nœuds pour lesquels cette expression renvoie $vrai$ : $result \leftarrow \{x \in result : pred_1(x) \wedge \cdots \wedge pred_n(x)\}$.

**P5.** [Terminaison] L'algorithme termine et renvoie $result$.

À l'intérieur d'un prédicat, le nœud courant par lequel le prédicat est évalué est accessible grâce à la fonction spéciale `current()`, ou plus simplement à travers la notation « point » (`.`).

**Exemple détaillé.** Ainsi, pour sélectionner toutes les interfaces de contrôle d'un composant[4], il suffit d'évaluer l'expression FPath suivante relativement à ce composant :

```
union(interface::component, interface::*[endswith(name(.), "-controller")])
```

Elle fonctionne de la façon suivante :

- La fonction prédéfinie `union()` prend un nombre quelconque de paramètres, qui doivent être des ensembles de nœuds, et renvoie leur union.

[4] La notion d'interface de contrôle n'est pas explicite dans le modèle Fractal, mais plutôt le résultat d'une convention. En pratique, les interfaces de contrôle ont un nom qui se termine par `"-controller"`, sauf pour l'interface `component` du noyau.

– La première sous-expression, `interface::component` est constituée d'un seul pas et n'a pas de prédicat. Elle sélectionne les nœuds interface reliés au composant de départ et dont le nom est `component`. Cette expression renvoie donc un singleton qui contient l'unique interface `component` du composant.
– La secondssse sous-expression est aussi un chemin a un seul pas qui suit l'axe `interface`. Puisqu'un test n'est pas assez puissant pour sélectionner les nœuds voulus, le test est `*`, qui conserve tous les nœuds, et le filtrage se fait dans les prédicats. Le seul prédicat présent ici vérifie que le nom du nœud courant (dans ce cas des interfaces) se termine bien par la chaîne `"controller"`.

Le résultat final est donc un ensemble de nœuds de type interfaces qui contient toutes les interfaces de contrôle du composant initial.

**Fonctions prédéfinies.** L'ensemble des fonctions standard permettant d'introspecter les différents nœuds et de manipuler les types de base est décrit en détail dans l'annexe A, page 3. En pratique, toutes les méthodes d'introspection pure des interfaces définies par Fractal ont leur équivalent dans la « bibliothèque standard », qui contient en plus quelques fonctions supplémentaires pour faciliter certaines tâches.

**Axes synthétiques.** En plus des axes de base pour la navigation qui correspondent directement aux arcs du graphe sous-jacent, FPath définit un ensemble d'axes dits *synthétiques* qui enrichissent le pouvoir d'expression du langage tout en conservant la garantie de terminaison des expressions. En effet, FPath ne permet de naviguer dans le graphe qu'à une distance fixe et définie par la longueur du chemin. Il est en particulier impossible avec les axes présentés jusqu'ici d'écrire une expression FPath qui sélectionne tous les sous-composants, directs ou indirects, d'un composant donné. Cependant, les contraintes posées par Fractal nous garantissent qu'une telle expression terminera toujours, puisque qu'il n'y a pas de cycles de composition et que le nombre de composants (et donc de nœuds) existant est fini. Pour permettre ce genre d'expressions, nous ajoutons donc les axes suivants :

– **descendant** et **ancestor** sélectionnent respectivement l'ensemble des sous-composants et des super-composants *directs et indirects* du composant de départ. Ces axes correspondent à la clôture récursive des axes child et parent.
– **sibling** sélectionne tous les composants qui se trouvent « au même niveau » que le composant de départ, c'est-à-dire qui sont sous-composants directs d'au moins un des parents directs du composant de départ. Il s'agit en fait d'un raccourci syntaxique pour `parent::*/child::*[.!=$c]` (où `$c` représente le composant de départ).
– **external-interface** et **internal-interface** sont des variantes de l'axe interface qui ne sélectionnent respectivement que les interfaces externes et internes du composant.

Enfin, il existe des variantes de certains axes qui incluent le nœud de départ dans l'ensemble résultat : **descendant-or-self**, **ancestor-or-self**, **child-or-self**, **parent-or-self** et **sibling-or-self**.

**Raccourcis syntaxiques.** Afin de rendre certaines expressions plus lisibles, FPath définit des notations spécifiques pour certains axes, qui ne nécessitent pas de séparateur `::` entre l'axe et le test :

– La notation `@` permet de sélectionner l'axe `attribute`. Ainsi, `@foo` représente l'attribut `foo` d'un composant (s'il existe), et `@*` l'ensemble des attributs d'un composant.
– De même, la notation `#` permet de sélectionner l'axe `binding`. `$c/\#service/component::*/@attr` sélectionne le paramètre de configuration `attr` du composant connecté à l'interface `service` de `$c`.

Enfin, dans beaucoup de cas (voir l'exemple précédent), un chemin doit utiliser le pas `component::*` pour passer d'un nœud de type interface à un attribut, ou réciproquement. FPath permet d'omettre ce pas pour rendre les expressions plus simples et plus lisibles. Puisque les types de nœuds en entrée ou en sortie d'un pas ne dépendent que de l'axe, qui est connu statiquement, FPath insère un pas `component::*` lorsque cela est nécessaire pour rendre le chemin correct. Ainsi, l'expression de l'exemple précédent peut encore être abrégée en `$c/#service/@attr`.

### 6.3.2 Types de données et expressions de base

Nous présentons maintenant le reste du langage, beaucoup plus classique, qui couvre les aspects du langage FPath qui ne sont pas spécifiques au domaine Fractal. La syntaxe concrète des différents littéraux et opérateurs est résumée dans le tableau 6.1.

**Nombres.** Les nombres manipulés par FPath correspondent aux nombres flottants à précision double de Java, et plus précisément au type `java.lang.Double`. La syntaxe des littéraux numériques est la même que celle de Java, à la différence près que tous les littéraux sont interprétés comme des flottants. FPath n'ayant ni valeur de type `null` ni mécanisme de gestion d'erreurs, les expressions numériques invalides ou qui doivent signaler une erreur (par exemple une division par 0) renvoie la valeur spéciale $NaN$ (*Not a Number*), équivalente à la valeur Java `Double.NaN`. Cette valeur spéciale n'a pas de syntaxe littérale, mais peut être obtenue par la fonction sans paramètre $NaN()$. Tout comme dans le cas de Java, la comparaison entre deux expressions dont la valeur est $NaN$ renvoie toujours *faux*. La seule façon de tester si une valeur numérique est bien $NaN$ est d'utiliser la fonction $isNaN(expr)$ qui évalue l'expression passée en paramètre et renvoie *vrai* si et seulement si sa valeur est $NaN$.

<table>
<tr><th colspan="2">Littéraux</th><th colspan="2">Arithmétique</th></tr>
<tr><td colspan="2">42, -3, 3.14, -2.718281828, 6.02e-23</td><td>$expr_1$ + $expr_2$</td><td>$expr_1$ - $expr_2$</td></tr>
<tr><td colspan="2">"", "FPath", "A\nmulti-line\nstring.", "Une chaîne \"entre guillemets\"."</td><td>$expr_1$ * $expr_2$</td><td>$expr_1$ div $expr_2$</td></tr>
<tr><th colspan="2">Comparaisons</th><th colspan="2">Logique</th></tr>
<tr><td>$expr_1$ = $expr_2$</td><td>$expr_1$ != $expr_2$</td><td colspan="2">not( $expr_1$ )</td></tr>
<tr><td>$expr_1$ < $expr_2$</td><td>$expr_1$ > $expr_2$</td><td colspan="2">$expr_1$ and $expr_2$</td></tr>
<tr><td>$expr_1$ <= $expr_2$</td><td>$expr_1$ >= $expr_2$</td><td colspan="2">$expr_1$ or $expr_2$</td></tr>
<tr><th colspan="2">Variables</th><th colspan="2">Fonctions</th></tr>
<tr><td colspan="2">$n, $a_string, $empty?, $current-state</td><td colspan="2">nom_fonction($expr_1$, $expr_2$, ..., $expr_n$)</td></tr>
</table>

TAB. 6.1 – Syntaxe concrète de FPath.

**Expressions arithmétiques.** Les opérateurs arithmétiques habituels pour les quatre opérations de base sont aussi supportés, avec les priorités habituelles (multiplication et division prioritaires sur l'addition et la soustraction). La syntaxe de la division est différente pour éviter les conflits avec les expressions de chemin qui utilisent le caractère `/` (*slash*) comme séparateur. Si une opération génère un dépassement de capacité (i.e. le résultat est trop grand ou trop petit pour être représenté par un `Double` Java), la valeur retournée par l'expression est $NaN$. De même, le résultat d'une division par 0 est $NaN$ mais ne produit pas d'erreur.

**Chaînes de caractères.** Les chaînes de caractères de FPath sont elles aussi modélisées sur celles de Java, et sont donc constantes (non mutables). Leur syntaxe est identique à celle de Java, y compris pour les caractères spéciaux `\n`, `\t`, etc. Aucun opérateur spécifique n'existe pour les chaînes, mais FPath définit un ensemble de fonctions standards permettant de les manipuler.

**Booléens.** Les valeurs booléennes *vrai* et *faux* n'ont pas de représentation littérale en FPath (pour éviter les conflits dans la grammaire), mais sont accessibles par les deux fonctions `true()` et `false()`.

**Combinateurs booléens.** FPath supporte les combinateurs logiques standards : conjonction, disjonction et négation, mais la négation est gérée par une fonction à un argument plutôt que par un opérateur. La conjonction est prioritaire sur la disjonction. Ainsi l'expression $expr_1$ `and` $expr_2$ `or` $expr_3$ est interprétée comme l'expression `(` $expr_1$ `and` $expr_2$ `)` `or` $expr_3$. Puisque toutes les expressions FPath sont par définition

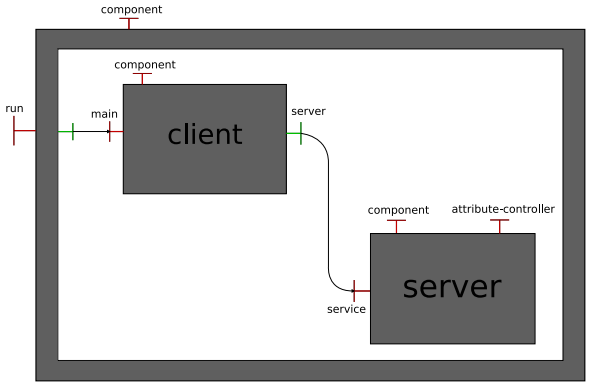


FIG. 6.2 – Un exemple d'application Fractal simple.

sans effet de bord, l'ordre d'évaluation des opérandes ou le fait qu'une implémentation « court-circuite » l'évaluation de certaines expressions n'a pas d'impact sur la sémantique des expressions.

**Comparaisons.** Les opérateurs de comparaison classiques sont aussi supportés. Les tests d'égalité et de différence fonctionnent pour tous les types de données. Dans le cas particulier des chaînes de caractères, le test d'égalité utilise la méthode `String.equals()` de Java, et teste donc l'égalité du contenu des chaînes. Ainsi, `"foo" = "foo"` est toujours *vrai*. Les autres comparaisons (`<`, `>`, etc.) ne fonctionnent qu'avec des nombres. Leur sémantique est exactement la même que celle de leurs équivalents Java.

**Variables.** Bien qu'il ne soit pas possible de définir de nouvelles variables dans une expression FPath, il est possible d'en référencer. Leur définition est à la charge du programme qui invoque l'évaluation d'une expression. Les variables FPath référencent des valeurs de n'importe quel type, c'est-à-dire que bien que les *valeurs* FPath soient typées, les *variables* ne le sont pas (typage dynamique). De plus, le langage et l'interface de programmation garantissent qu'une variable a toujours une valeur ; il n'y a pas d'équivalent de `null`.

Le nom d'une variable peut être n'importe quel identifiant, constitué de lettres, de nombres, et des caractères `-?:_` et dont le premier caractère est une lettre. FPath est sensible à la casse : les identifiants $x$ et $X$ sont considérés comme distincts. Pour référencer une variable et obtenir sa valeur dans une expression, il suffit de préfixer son nom par un caractère `$` (dollar).

**Invocation de fonctions.** Nous l'avons déjà évoqué dans les descriptions précédentes sans le décrire clairement, mais FPath permet d'invoquer des fonctions, qui doivent être *pures* au sens où elle ne modifient pas le système. Les noms de fonctions suivent les mêmes règles que ceux des variables, mais fonctions et variables sont définies dans des espaces de noms différents. Il est donc possible d'avoir à la fois une variable `f` et une fonction `f` sans aucun risque de collision. L'invocation des fonctions se fait tout simplement en utilisant la syntaxe habituelle. Le système évalue les expressions passées en paramètres, dans un ordre non défini, et invoque ensuite la fonction elle-même en lui passant en paramètre les valeurs obtenues. Une fonction renvoie toujours une valeur.

**Groupement d'expressions.** Enfin, comme certains des exemples plus haut le montrent, il est possible de grouper des expressions entre parenthèses pour modifier leur ordre d'évaluation par rapport à celui déterminé par la priorité des opérateurs.

### 6.3.3 Quelques exemples d'expressions FPath

Ces exemples sont décrits dans le contexte de l'application exemple de la figure 6.2.

**Navigation simple**

**Expression :** `child::client/binding::server/parent::*`

**Explication :** Trouve le(s) fils de `root` nommé(s) `client`, puis suit la connexion de l'interface `server` de ceux-ci. Le résultat intermédiaire est un ensemble (ici in singleton) d'interfaces. Enfin, suit l'axe `parent` de ces interfaces (en fait du composant correspondant). Le résultat final est ici un singleton contenant le composant `root`.

**Quels sont les composants configurables de mon application ?**

**Expression :** `descendant-or-self::*[@*]`

**Explication :** Recherche récursivement les composants pour lesquels l'expression `@*` (équivalente à `attribute::*`) est vraie, i.e. les composants dont l'ensemble des attributs est non-vide. La valeur renvoyée est un ensemble contenant uniquement le composant `server`. La variante suivante permet d'obtenir directement les attributs configurables eux-mêmes : `descendant-or-self::*/@*`.

**Le composant courant a-t-il des interfaces requises pas encore connectées ?**

**Expression :** `count(interface::*[required(.) and not(bound(.))]) > 0`

**Explication :** Dans les prédicats, le point (`.`) représente le nœud courant auquel est appliqué le test.

**Étant donnée une interface à connecter (par exemple renvoyée par la requête précédente), quels sont les candidats compatibles ?**

**Expression :** `sibling::*/interface::*[provided(.)][not(bound(.))][subtype(., $itf)]`

**Explication :** La variable `$itf` représente l'interface à connecter.

**Quels sont les composants partagés ?**

**Expression :** `descendant-or-self::*[count(parent::*) > 1]`

**Explication :** Évaluer l'expression à partir de la racine de l'application.

**Après navigation à partir de la racine, obtenir l'attribut de configuration `header` du composant serveur.**

**Expression :** `client/binding::s/@header`

**Explication :** Pour comparaison, voici le code Fractal générique équivalent :

```
try {
  ContentController cc = Fractal.getContentController(root);
  Object[] children = cc.getFcSubComponents();
  for (int i = 0; i < children.length; i++) {
    Component kid = (Component) children[i];
    String name = null;
    try {
      NameController nc = Fractal.getNameController(kid);
      name = nc.getFcName();
    } catch (NoSuchInterfaceException nsie) {
      name = "";
    }
    if (name.equals("client")) {
      try {
        BindingController bc = Fractal.getBindingController(kid);
        Interface itf = (Interface) bc.lookupFc("s");
        if (itf != null) {
          Component other = itf.getFcItfOwner();
          AttributeController ac= Fractal.getAttributeController(other);
          Class klass = ac.getClass();
          try {
            Method meth = null;
            try {
              meth = klass.getMethod("getHeader", null);
            } catch (NoSuchMethodException nime) {
```

```
            meth = klass.getMethod("isHeader", null);
          }
          if (meth != null) {
            try {
              Object value = meth.invoke(ac, null);
              System.out.println(value);
            } catch (Exception e) { /* ignore */
            }
          } catch (Exception e) { /* ignore */ }
        }
      } catch (NoSuchInterfaceException nsie) { /* ignore */ }
    }
  }
} catch (NoSuchInterfaceException nsie) {
  // ignore
}
```

### 6.3.4 Interface de programmation

Du point de vue du programmeur Java, FPath se présente sous la forme d'une interface de programmation très simple, essentiellement constituée de trois classes :

- `FPath`[5] est une classe utilitaire qui permet d'analyser (*parser*) une chaîne de caractères représentant une expression FPath et d'obtenir en retour un objet de type `Expression`.
- `FPathInterpreter` représente un interprète FPath, associé à un *contexte d'évaluation* qui regroupe les valeurs des variables et les définitions de fonctions connues par l'interprète, et permet de manipuler ces derniers.
- Enfin, l'interface `FractalValueFactory` permet d'encapsuler des objets Java en leur équivalent FPath, qu'il s'agisse de types de base (chaînes, nombres et booléens) ou de types spécifiques à Fractal (composants, interfaces et attributs). Chaque interprète FPath – instance de `FPathInterpreter` – permet d'obtenir une *Fabrique* [Gamma et al., 1994] de valeurs compatibles avec son implémentation.

La procédure typique pour évaluer une ou plusieurs expressions FPath dans le contexte d'une application Fractal est la suivante :

1. Obtenir un interprète et une fabrique de valeurs :

   ```
   FPathInterpreter inter = new FPathInterpreter();
   FractalValueFactory fact = inter.getValueFactory();
   ```

   L'interprète ainsi créé est prêt à être utilisé et contient les définitions de toutes les fonctions standards, mais d'aucune variable. Les valeurs créées par la fabrique ne doivent être utilisées qu'avec l'interprète qui l'a créée. Inversement, un interprète donné ne peut utiliser que des valeurs créées par une fabrique obtenue grâce à lui.

2. (*Optionnel*) Définir de nouvelles variables et/ou fonctions qui seront alors disponibles pendant l'évaluation des expressions FPath :

   ```
   inter.define("target-component", fact.createComponentNode(aComponent));
   inter.define(max", fact.createNumber(5));
   inter.loadFunction("function1", MyNewFunction.class);
   inter.loadDefinitions(aReader);
   ```

   Les définitions de fonctions utilisent soit des classes Java (qui implémentent l'interface `Function`), soit des définitions FScript (décrites dans la section 6.4 de ce chapitre) lisibles par `aReader`.

3. Obtenir une `Expression` à partir de sa définition textuelle, l'évaluer dans le contexte d'un nœud de départ (ici un nœud composant qui doit d'abord être créé), puis récupérer la ou les valeurs réelles encapsulées afin de pouvoir les utiliser :

[5] Toutes les classes et interfaces décrites dans ce chapitre font partie du paquetage Java `org.obasco.fractal.fscript` ou d'un de ses sous-paquetages, même pour celles qui ne concernent que le sous-langage FPath.

```
Expression expr = FPath.parse("descendant-or-self::*[size(./child::*) > $max]");
Value val = interp.evaluate(expr, fact.createComponentNode(rootComponent));
Set components = new HashSet();
for (Iterator iter = ((NodeSet) val).iterator(); iter.hasNext(); ) {
    ComponentNode node = (ComponentNode) iter.next();
    components.add(node.getComponent());
}
```

L'ensemble `components` contient alors tous les sous-composants (`Component`) directs ou indirects de `rootComponent` qui contiennent plus de `$size-limit` (5) sous-composants directs.

## 6.4 Reconfiguration de l'architecture avec FScript

Nous décrivons maintenant le langage FScript dans sa globalité. Par rapport à FPath, FScript ajoute la possibilité de définir de nouvelles *fonctions* (utilisables dans des expressions FPath), et surtout des *actions de reconfiguration* qui, contrairement aux fonctions, sont capables de *modifier* l'architecture des composants Fractal. Pour cela, FScript supporte un certain nombre de structures de contrôle, et surtout enrichit la « bibliothèque standard » de FPath avec un ensemble d'*actions primitives* qui permettent d'agir sur les composants Fractal.

### 6.4.1 Structure générale : définitions de fonctions et d'actions

Un programme FScript est constitué d'un ensemble de définitions de *fonctions* et d'*actions*. La différence entre les deux est que les fonctions sont strictement sans effet de bord sur l'architecture Fractal : seules les actions peuvent invoquer d'autres actions (primitives ou non). Partout où le langage accepte des expressions FPath, celles-ci peuvent invoquer les *fonctions* définies par l'utilisateur (en plus de celles prédéfinies), mais ne peuvent invoquer d'*actions*. Cette distinction permet d'offrir certaines garanties sur le comportement des scripts de reconfiguration : certaines constructions du langage sont ainsi garanties sans effet de bord sur l'architecture de l'application cible.

Dans le reste de ce chapitre, nous parlerons de *procédure* pour désigner à la fois les fonctions et actions puisque beaucoup de leurs caractéristiques sont communes. Afin de garantir la terminaison des procédures, *les définitions récursives* – directes ou indirectes – *sont interdites*.

La syntaxe abstraite complète d'un programme FScript est la suivante (notons que la production `Expression` de cette grammaire correspond au sous-langage FPath) :

```
Program ::= Definition+
Definition ::= FunctionDefinition | ActionDefinition
FunctionDefinition ::= 'function' Name '(' Arguments? ')' '=' Statement
ActionDefinition ::= 'action' Name '(' Arguments? ')' '=' Statement
Arguments ::= Name | Name ',' Arguments
Statement ::= Assignement
            | IfStatement
            | ForeachStatement
            | ReturnStatement
            | BlockStatement
            | CallStatement
Assignement ::= Name ':=' Expression ';'
IfStatement ::= 'if' Expression 'then' Statement
              | 'if' Expression 'then' Statement 'else' Statement
ForeachStatement ::= 'foreach' Name 'in' Expression 'do' Statement
ReturnStatement ::= 'return' Expression ';'
BlockStatement ::= '{' Statement+ '}'
CallStatement ::= CallExpression ';'
```

```
  // Expressions FPath
  Expression ::= LitteralValue
               | VariableReference
               | BooleanExpression
               | ArithmeticExpression
               | PathExpression
               | CallExpression
  LitteralValue ::= NumberLitteral | StringLitteral
  VariableReference ::= '$' Name
  BooleanExpression ::= Expression 'or' Expression
                      | Expression 'and' Expression
  ArithmeticExpression ::= Expression '+' Expression
                         | Expression '-' Expression
                         | Expression '*' Expression
                         | Expression 'div' Expression
  PathExpression ::= (VariableReference | CallExpression) Step+
  Step ::= '/' Name '::' (Name | '*') ('[' Expression ']')*
  CallExpression ::= Name '(' Parameters? ')'
  Parameters ::= Expression | Expression ',' Parameters
```

Le corps de la définition d'une procédure est constitué d'une séquence d'instructions ou dans le cas de la définition d'une action d'invocations d'autres actions. Les instructions d'affectation, de retour explicite ou d'invocation d'action doivent être terminées par un point-virgule ( `;`). Des commentaires peuvent être insérés dans un programme FScript en utilisant les deux types de syntaxes supportés par C++ (`//` et `/* ... */`).

Voici deux exemples complets de définitions FScript. La première est une fonction qui teste la compatibilité de deux interfaces, et pourrait être utilisée dans n'importe quelle expression FPath (y compris par exemple pour définir des contraintes architecturales grâce à notre extension). La seconde définition est celle d'une action générique qui tente de connecter automatiquement les interfaces requises d'un composant donné. Elle utilise pour cela l'action prédéfinie `bind()`, qui serait interdite dans la définition d'une simple fonction.

```
// Renvoie vrai ssi les deux interfaces sont de type compatible.
function compatible?(client-itf, server-itf) = {
  if (client($client-itf) and server($server-itf)) then {
    client-type := type($client-itf);
    server-type := type($server-itf);
    return subtype($client-type, $server-type);
  } else {
    return false();
  }
}

/* Connecte toutes les interfaces externes clientes de $comp à une
 * interface serveur compatible, si une telle interface est visible.
 * Si $all est faux, ne tente de connecter que les interfaces clientes
 * requises (mandatory), si $all est vrai, tente de connecter toutes
 * les interfaces clientes. S'il existe plusieurs interfaces serveur
 * possibles, en choisit une arbitrairement.
 *
 * Renvoie vrai si et seulement si toutes les interfaces dont la
 * connexion a été demandée ont pu être connectées.
 */
```

```
action auto-bind(comp, all) = {
  if ($all) then {
    clients := $comp/external-interface::*[client(.)][not(bound(.))];
  } else {
    clients := $comp/external-interface::*[client(.)][mandatory(.)][not(bound(.))];
  }
  complete := true();
  foreach itf in $clients do {
    candidates := $comp/sibling::*/external-interface::*[compatible?($itf, .)];
    if (not(empty?($candidates))) then
      bind($itf, one-of($candidates));
    else
      complete := false();
  }
  return $complete;
}
```

### 6.4.2 Affectations, portée et durée de vie des variables

FScript supporte trois catégories de variables, qui diffèrent par la façon dont elles sont définies, leur portée et leur durée de vie. Dans tous les cas, la valeur d'une variable nommée `foo` est accessible par l'expression `$foo`, et peut être modifiée par une affectation de la forme :

```
foo := <expression fpath>;
```

Les trois catégories sont :

1. Les *variables globales*, définies grâce aux interfaces de programmation permettant de créer et d'utiliser un interprète FScript (de façon similaire aux interprètes FPath). Un programme FScript ne peut pas créer de telles variables, mais peut utiliser et modifier leur valeur. Elles sont utiles pour paramétrer des programmes FScript et pour permettre de conserver un état entre plusieurs invocations successives de procédures FScript. Ces variables sont visibles de toutes les procédures exécutées dans l'interprète qui les définit, à moins qu'une variable de même nom ne la masque. Leur durée de vie, déterminée par le code client qui gère l'interprète, est indépendante de celle des procédures qui les utilisent.
2. Les *variables locales* sont définies à l'intérieur d'une procédure, et ne sont visibles que dans le corps de celle-ci. Une nouvelle variable locale est implicitement créée la première fois qu'une valeur lui est affectée. Il n'est donc pas possible de masquer une variable globale par une variable locale, puisqu'une expression de la forme

   ```
   foo := <expression>;
   ```

   évaluée dans un contexte où il existe une variable globale `$foo` est toujours considérée comme une modification de la variable globale. Une fois définie, une variable locale est visible dans tout le reste de la procédure : les différentes structures de contrôle – y compris les blocs – n'introduisent pas de nouvelle portée. La durée de vie d'une variable locale est celle de l'invocation de la procédure qui l'a créée.
3. Les *paramètres* passés lors de l'invocation d'une procédure sont visibles dans le corps de celle-ci comme une variable locale dont le nom est celui de l'argument correspondant dans la définition de la procédure. La valeur d'un paramètre est initialement celle de l'expression passée lors de l'invocation, mais le corps de la procédure peut affecter une nouvelle valeur à la variable correspondante. En dehors de leur valeur initiale définie automatiquement, et du fait qu'ils peuvent masquer des variables globales du même nom, les paramètres se comportent exactement comme des variables locales.

La fonction prédéfinie `defined(name)` prend en paramètre une chaîne de caractères et renvoie *vrai* si et seulement si il existe au moment de son invocation une variable visible sous ce nom, quel que soit son type.

### 6.4.3 Structures de contrôle

#### Blocs d'instructions

Un bloc regroupe tout simplement une séquence finie non vide d'instructions, qui sont exécutées dans l'ordre jusqu'à l'exécution éventuelle d'un retour explicite (`return`). Les blocs sont délimités par des accolades `{` et `}`. Contrairement à certains langages, les blocs FScript n'introduisent pas de nouvelle portée de variable. Ainsi, étant donnée la définition suivante :

```
function f(bar) = {
  foo := "foo";
  if ($bar) then {
    foo := "bar";
  }
  return $foo;
}
```

l'invocation `f(true())` renvoie la chaîne `"bar"`.

#### Alternative

L'alternative (*si/sinon*) permet d'exécuter des instructions différentes suivant la valeur booléenne d'une expression. La syntaxe est la suivante :

```
if (<test>) then <instruction1>
if (<test>) then <instruction1> else <instruction2>
```

L'expression `<test>` est tout d'abord évaluée, et sa valeur de retour est convertie automatiquement en booléen (par un appel implicite à la fonction prédéfinie `boolean()`). Si le résultat est *vrai*, alors `<instruction1>` est exécutée. Si le résultat est *faux*, alors `<instruction2>` est exécutée si elle est présente.

La syntaxe concrète de FScript permet d'écrire des « cascades » d'alternatives, qui sont alors considérées comme imbriquées. Ainsi, l'exemple suivant :

```
if (<test1>) then
  <instruction1>
else if (<test2>) then
  <instruction2>
else if (<test3>) then
  <instruction3>
else
  <instruction4>
```

est interprété comme en C ou en Java :

```
if (<test1>) then
  <instruction1>
else {
  if (<test2>) then
    <instruction2>
  else {
```

```
    if (<test3>) then
      <instruction3>
    else {
      <instruction4>
    }
  }
}
```

### Itération

L'instruction d'itération de FScript permet d'appliquer une instruction de façon répétée à un ensemble d'éléments. Cette fonctionnalité est nécessaire pour pouvoir écrire des scripts de reconfiguration suffisamment généraux tirant partie des capacités du langage FPath (qui permet justement de sélectionner un ensemble de nœuds). Plusieurs possibilités existent pour introduire cette fonctionnalité dans le langage, comme par exemple la récursion ou une instruction de boucle généralisée (`loop` ou `while`). Cependant, ces deux exemples sont trop généraux et puissants, et leur introduction dans le langage rendrait celui-ci Turing-complet, ce qui irait à l'encontre de notre objectif d'un langage dédié capable d'être analysé. En particulier, ces deux constructions empêcheraient de pouvoir garantir la terminaison des reconfigurations.

L'instruction d'itération est donc conçue pour offrir suffisamment de pouvoir d'expression aux programmeurs de reconfigurations tout en conservant la garantie de terminaison. En pratique, cela signifie que la forme même de l'instruction ne permet de répéter une instruction qu'un nombre fini de fois.

Sa syntaxe est la suivante :

```
foreach <var> in <nodeset> do <instruction>
```

L'expression FPath `<nodeset>` est tout d'abord évaluée, et doit renvoyer un ensemble de nœuds, éventuellement vide. L'instruction `<instruction>` est alors exécutée une fois pour chacun des éléments de cet ensemble, dans un ordre non défini, la variable `<var>` ayant à chaque itération la valeur de l'élément correspondant.

Quelques exemples :

```
// Retire tous les sous-composants directs du composite $c.
foreach kid in $c/child::* do
  remove($c, $kid);

// Déconnecte toutes les connexions de $c et le retire
// de tous ses parents.
foreach itf in $c/internal-interface::* do
  if (bound($itf)) then unbind($itf);
foreach father in $c/parent::* do
  remove($father, $c);
```

### Retour explicite

La dernière construction du langage est le mot-clé `return`, qui permet d'interrompre immédiatement l'exécution d'une procédure et de renvoyer une valeur à l'appelant (qui peut être soit une autre procédure, soit un programme Java). Sa syntaxe est tout simplement :

```
return <expression>;
```

où `<expression>` est une expression FPath quelconque qui est évaluée et dont la valeur est celle de l'invocation.

Tous les chemins d'exécution possibles de toutes les procédures doivent se terminer par une instruction `return`. De plus, cette instruction doit être la dernière du chemin d'exécution. Si ce n'est pas le cas, l'analyseur syntaxique de FScript signale une erreur indiquant qu'une portion de code est inatteignable.

### 6.4.4 Actions de reconfiguration primitives

FScript inclut une « bibliothèque standard » d'actions de reconfigurations primitives utilisables pour modifier les composants Fractal d'une application. La plupart de ces actions correspondent directement à des méthodes des interfaces de contrôle de Fractal, les noms et paramètres étant adaptés à FScript.

Syntaxiquement, la forme de ces actions est la même que pour les fonctions prédéfinies dans FPath. Cependant, l'implémentation de FScript sait que ces opérations sont spéciales et qu'elles peuvent modifier l'architecture.

Nous ne donnons ici qu'une présentation relativement rapide de ces actions primitives.

#### Paramétrage des composants

La modification des paramètres de configuration des composants se fait grâce à l'action primitive `set-value()` :

```
set-value(attribute, value)
```

Le premier paramètre, `attribute`, doit être un nœud attribut, ou un ensemble de tels nœuds (éventuellement vide). `value` est une expression FPath dont la valeur doit être d'un des types de base (nombre, chaîne ou booléen). L'exécution de cette action modifie le ou les paramètre(s) de configuration désignés par `attribute` en leur affectant la nouvelle valeur `value`. La valeur de retour est un booléen qui indique si un paramètre a été effectivement modifié, i.e. l'action renvoie *false* uniquement si `attribute` est un ensemble vide ou si tous les paramètres avaient déjà la valeur `value`.

Fractal n'a pas de méthode strictement équivalente, puisque les paramètres de configuration n'y sont pas réifiés mais n'existent que grâce à des conventions de nommage. En pratique, le code FScript

```
set-value($c/@foo, 42);
```

correspond au (pseudo-)code Java

```
c.getFcInterface("attribute-controller").setFoo(42);
```

Pour que cette action soit valide, c'est-à-dire pour qu'elle s'exécute sans erreur, il faut en plus des contraintes sur les types des paramètres, que les attributs désignés par `attribute` existent et que leur type soit compatible avec celui de `value`. La première condition est garantie par construction puisque FPath ne permet de désigner que des nœuds qui existent réellement. La seconde condition ne peut être vérifiée qu'à l'exécution. Si le type Java du paramètre désigné est *booléen* (primitif ou `Boolean`), nombre (primitif ou `Number`) ou chaîne de caractère (`String`), FScript convertit automatiquement la valeur `value` dans le type correspondant en utilisant les fonctions FPath `boolean()`, `number()` ou `string()`. Malgré cela, des erreurs peuvent toujours se produire si le type Java est un type objet non supporté par FScript. Dans ce cas, l'exécution de l'action échoue.

#### Renommage d'un composant

L'action primitive `set-name()` permet de modifier le nom sous lequel un composant est accessible (`name-controller`) :

```
set-name(component, name)
```

L'argument `component` doit désigner un nœud composant, ou éventuellement un ensemble de nœuds composants, et l'argument `name` doit être une chaîne de caractères. Puisque le nom d'un composant n'a qu'une valeur indicative, il n'y a pas d'autres contraintes de validité pour cette action, qui ne peut jamais échouer – tant que les types des arguments sont respectés. Tout comme pour l'action précédente `set-value()`, la valeur de retour de `set-name()` est un booléen qui indique si au moins un nom de composant a été effectivement modifié par l'opération. Si l'un des composants désignés ne possède pas d'interface `name-controller`, et donc pas de nom, la méthode n'a aucun effet mais ne génère pas d'erreur.

Du point de vue de l'API Fractal, cette action correspond à la méthode `setName()` de l'interface `NameController`. Ainsi, le code FScript suivant

```
set-name($c, "foo");
```

correspond au code Java

```
((NameController) c.getFcInterface("name-controller")).setFcName("foo");
```

### Ajout et retrait de sous-composants

La modification du contenu des composites (`content-controller`) – et donc de l'aspect hiérarchique de l'architecture – se fait grâce aux deux actions primitives `add()` et `remove()` :

```
add(composite, sub-component)
remove(composite, sub-component)}
```

`add()` permet d'ajouter un nouveau sous-composant à un composite, et `remove()` permet d'en retirer un. Dans les deux cas, les deux paramètres `composite` et `sub-component` doivent désigner des nœuds composants, ou bien des ensembles de nœuds composants. Plus spécifiquement les composants désignés par `composite` doivent être des composites, c'est-à-dire qu'ils doivent fournir l'interface de contrôle `content-controller`. L'effet de l'action `add()` (resp. `remove()`) est d'ajouter à (resp. de retirer de) l'ensemble des composites désignés par `composite` tous les sous-composants désignés par `sub-component`. L'action retourne un booléen qui est *vrai* si et seulement si la composition d'au moins un composite a été effectivement modifiée par l'opération. En plus d'éventuels problèmes de typage des arguments, l'action `add()` échoue si l'ajout d'un des sous-composants dans un des composites crée un cycle de composition. Si un composant qui doit être ajouté à (resp. retiré de) un composite est déjà (resp. n'est pas) un sous-composant direct de ce dernier, alors l'action `add()` (resp. `remove()`) n'a aucun effet, mais n'échoue pas.

Dans l'implémentation de Fractal que nous utilisons, un composite doit être stoppé (état `STOPPED`) avant de pouvoir manipuler son contenu. Afin de faciliter l'écriture de reconfigurations, les actions `add()` et `remove()` se chargent automatiquement de mettre le composite dans l'état approprié s'il n'y est pas déjà, avant d'effectuer l'opération. FScript garantit qu'à la fin d'une reconfiguration, tous les composants dont l'état a été ainsi modifié implicitement retrouvent leur état initial[6], à moins qu'une action ultérieure ne l'ait modifié explicitement.

Bien que la spécification Fractal ne définisse les opérations correspondantes que de façon très générale afin de permettre une plus grande liberté aux implémentations, celle que nous utilisons (Julia) définit complètement l'effet de ces deux opérations. Si le contenu initial d'un composite `p` est `c`, alors après l'ajout à (resp. le retrait de) ce composite du composant `k`, le contenu de `p` est égal à $\texttt{c} \cup \texttt{k}$ (resp. $\texttt{c} \setminus \texttt{k}$).

La correspondance entre ces deux actions et les APIs Fractal est la suivante :

```
add($parent, $child);
remove($parent, $child);
```

correspondent à :

```
ContentController cc = (ContentController) parent.getFcInterface("content-controller");
cc.addFcSubComponent(child);
cc.removeFcSubComponent(child);
```

### Manipulation des connexions

Les actions primitives `bind()` et `unbind()` permettent respectivement de connecter ou de déconnecter des interfaces de composants :

```
bind(client-itf, server-itf)
unbind(client-itf)
```

[6] Ce retour à la normale ne se fait pas forcément immédiatement après l'application de l'action mais peut être différé – au choix de l'implémentation – jusqu'à la fin de la reconfiguration principale.

Dans les deux cas, le paramètre `client-itf` peut être soit une interface cliente, soit un ensemble de telles interfaces. Le second paramètre de `bind()`, `server-itf` doit désigner une interface serveur.

L'action `bind()` établit une connexion entre chacune des interfaces désignées par `client-itf` et l'unique interface `server-itf`. Pour pouvoir établir une telle connexion, les conditions suivantes doivent être remplies :

1. L'interface client n'est pas déjà connectée.
2. Les deux interfaces à connecter sont compatibles, c'est-à-dire que le type Java de l'interface serveur est un sous-type du type Java de l'interface client.
3. Les deux interfaces à connecter appartiennent à des composants distincts qui doivent posséder un parent direct en commun.

Si toutes ces conditions ne sont pas respectées, l'action `bind()` échoue. Lorsqu'elle n'échoue pas, `bind()` renvoie toujours *vrai*.

L'action `unbind()` déconnecte les interfaces désignées par `client-itf` des interfaces serveur auxquelles chacune d'elles est connectée. Elle est plus souple que `bind()` au niveau des pré-conditions puisque si une interface cliente n'est pas connectée, l'action n'a aucun effet et se contente de renvoyer *faux*. Cette action n'échoue donc jamais tant que son argument désigne bien des interfaces clientes. Si elle réussit, `unbind()` renvoie *vrai*.

Comme dans le cas des actions `add()` et `remove()`, l'implémentation de Fractal que nous utilisons impose qu'un composant *client* doit être stoppé (état `STOPPED`) avant de pouvoir manipuler ses interfaces ; le composant auquel appartient l'interface serveur (dé-)connectée peut être démarré. Les opérations `bind()` et `unbind()` se chargent automatiquement de mettre le composite dans l'état approprié s'il n'y est pas déjà, avant d'effectuer l'opération, et FScript garantit qu'à la fin d'une reconfiguration, tous les composants dont l'état a été ainsi modifié implicitement retrouvent leur état initial, à moins qu'une action ultérieure ne l'ait modifié explicitement. Notons que ce redémarrage peut faire échouer la reconfiguration globale même si toutes les actions primitives ont été effectuées sans erreur dans le cas où une interface cliente obligatoire a été déconnectée ; le composant auquel elle appartient ne peut plus être redémarré car il se trouve alors dans un état *inconsistant*.

### Gestion du cycle de vie

Le cycle de vie des composants peut être manipulé explicitement par les actions primitives `start()` et `stop()` :

```
start(component)
stop(component)
```

Dans les deux cas, le paramètre `component` doit désigner un composant ou un ensemble de composants. L'action `start()` s'assure que les composants en question sont dans l'état `STARTED`, en les démarrant si nécessaire. Inversement, l'action `stop()` s'assure que les composants sont dans l'état `STOPPED`. L'action renvoie *vrai* si l'état d'au moins un composant a été effectivement modifié, et *faux* sinon. Dans l'implémentation que nous utilisons, tous les composants fournissent l'interface `lifecycle-controller` et ont le même cycle de vie ; si le type du paramètre `component` est respecté, ces deux actions n'échouent donc jamais.

Du point de vue de l'API Fractal, ces deux actions

```
start(c);
stop(c);
```

correspondent au code Java

```
((LifeCycleController) c.getFcInterface("lifecycle-controller")).startFc();
((LifeCycleController) c.getFcInterface("lifecycle-controller")).stopFc();
```

### Manipulation du lien méta

Le lien méta entre un composant de base et un méta-composant introduit grâce à notre extension du modèle Fractal peut être manipulé grâce à deux actions primitives :

```
set-meta(base-component, meta-component)
unset-meta(base-component)
```

L'action `set-meta()` est similaire à `bind`, mais établit une connexion entre un ou plusieurs composants de base `base-component` et un unique composant méta `meta-component`. Tous les composants désignés par `base-component` doivent posséder l'interface de contrôle `metalink-controller`, et le composant méta `meta-component` doit avoir une interface serveur de type `MessageHandler` disponible sous le nom `message-handler`. De plus, le composant de base ne doit pas déjà posséder de méta-composant. Si toutes ces conditions ne sont pas respectées, l'action `set-meta()` échoue. Sinon, l'action réussie et retourne *vrai*.

L'action `unset-meta()` est elle similaire à `unbind()` et supprime la connexion entre les composants de base et leurs méta-composants. Tout comme `unbind()`, les pré-conditions de cette action sont plus souples : l'action n'échoue pas si le composant de base n'a pas de méta-composant voire pas d'interface `metalink-controller`, mais se contente de ne rien faire à part renvoyer *faux*. Lorsqu'elle réalise effectivement au moins une déconnexion, cette action renvoie `vrai`

### Manipulation des politiques d'adaptation

Deux actions d'adaptation permettent de manipuler directement les politiques d'adaptation elles-mêmes, ou plus précisément leurs associations avec les composants de l'application :

```
attach(component, policy)
detach(component, policy)
```

Dans les deux cas, `component` est le composant cible, qui doit implémenter l'interface `adaptation--controller` de SAFRAN, et `policy` est le nom d'uune politique d'adaptation. L'action `attach()` ajoute `policy` à la liste des politiques d'adaptation qui s'appliquent à `component`. Elle échoue si `policy` est une politique invalide ou qui ne peut pas s'appliquer à `component`. `detach()` retire simplement `policy` de la liste des politiques de `component`.

### Création de composant

La dernière action primitive standard supportée par FScript est celle qui permet de créer de nouveaux composants :

```
create(template-name)
```

L'argument `template-name` doit être une chaîne de caractère qui désigne un descripteur de composant déclaré en utilisant l'ADL Fractal. Ce descripteur doit être visible de l'implémentation de Fractal utilisée par FScript. Cela revient typiquement à rendre accessibles les fichiers `.fractal` contenant les descripteurs de composants au système de chargement de classes et de ressources de Java (`CLASSPATH`). Si aucun descripteur de composant n'est visible sous le nom désigné, l'action échoue. Si des erreurs se produisent pendant l'instanciation (descripteurs de sous-composants ou classes Java non accessibles ou invalides par exemple), l'action échoue. Sinon, FScript se charge de toutes les étapes de chargement et d'instanciation du composant, et l'action `create()` renvoie un nœud qui désigne le composant nouvellement créé, initialement dans l'état `STOPPED`.

Notons que le nouveau composant n'étant lié au aucune autre et n'étant contenu dans aucun composite, il est important que le code FScript qui crée un nouveau composant récupère dans une variable ou utilise immédiatement la valeur renvoyée par `create()`. Sinon, ce nouveau composant n'est plus accessible par aucun moyen au code FScript.

## 6.5 Description de l'implémentation

Dans cette section, nous décrivons les détails concrets de l'implémentation actuelle du langage dédié FScript. Nous donnons tout d'abord un aperçu des interfaces de programmation qui permettent de charger et d'invoquer des scripts FScript dans un programme Java, puis nous décrivons le modèle d'exécution de l'interprète FScript qui lui permet d'offrir les garanties de consistance que nous avons identifiées dans la section 6.2.3.

### 6.5.1 Interface de programmation

L'interface de programmation de FScript est quasi-identique à celle de FPath. La seule différence importante est qu'en plus de permettre la définition de variables, un interprète FScript est capable de charger des définitions de nouvelles fonctions ou actions à partir d'un fichier texte.

Les classes `FPath` et `FPathInterpreter` sont remplacées par leurs équivalents `FScript` et `FScript-Interpreter` :

- `FScript` est une classe utilitaire qui permet d'analyser (*parser*) une chaîne de caractères représentant une expression FPath ou une instruction FScript et d'obtenir en retour un objet de type `Expression` ou `Statement`, respectivement.
- `FScriptInterpreter` représente un interprète FScript, associé à un *contexte d'évaluation* qui regroupe les valeurs des variables et les définitions des procédures connues par l'interprète.
- Enfin, l'interface `FractalValueFactory` est la même que celle décrite pour FPath, et permet d'encapsuler des objets Java en leur équivalent FScript.

La procédure typique pour utiliser FScript dans le contexte d'une application Fractal est la suivante :

```
FScriptInterpreter inter = new FScriptInterpreter();
FractalValueFactory fact = inter.getValueFactory();
// Définition de variables
inter.define("c", fact.createComponentNode(aComponent));
// Définition de nouvelles fonctions et actions
inter.loadFunction("function1", MyNewFunction.class);
inter.loadFunction("action1", MyNewAction.class);
inter.loadDefinitions(aReader);
// Exécution d'une instruction
Statement stat = FScript.parseStatement("my-reconfiguration($c)");
Value val = interp.execute(stat);
```

Les actions primitives décrites dans la section 6.4.4 sont prédéfinies et toujours disponibles dans FScript. Cependant, puisque le modèle Fractal est extensible et permet d'ajouter de nouvelles interfaces de contrôle, l'implémentation de FScript est aussi extensible et permet de définir facilement de nouvelles procédures primitives afin de rendre ces extensions utilisables par des programmes FScript.

Jusqu'ici, nous n'avons pas indiqué comment sont gérées les erreurs, qui peuvent toujours se produire lorsque l'on reconfigure une application en cours d'exécution. Le langage lui-même ne contient d'ailleurs aucun moyen pour signaler ou gérer des erreurs explicitement. Nous avons fait le choix de déléguer la détection et la gestion des erreurs à l'*implémentation* du langage. La seule contrainte au niveau de la spécification du langage est que toute implémentation du langage FScript doit garantir les critères de consistance décrits dans la section 6.2.3. Par rapport à une spécification plus formelle des critères de validité d'un programme FScript, cette approche a l'avantage d'offrir une plus grande souplesse aux implémentations quant aux stratégies à employer pour offrir ces garanties, sans compromettre la « sûreté » du langage. Du moment qu'elle garantit les critères de consistance des reconfigurations – la seule chose réellement importante – chaque implémentation est libre d'utiliser les techniques qui lui semblent les plus appropriées : système de type, analyses statiques plus ou moins complexes ou vérification dynamique par exemple.

La section suivante décrit justement comment l'implémentation actuelle garantit la consistance des reconfigurations.

### 6.5.2 Modèle d'exécution

Nous décrivons maintenant le modèle d'exécution utilisé par l'implémentation actuelle de FScript pour garantir les critères de consistances que nous avions identifiés concernant les reconfigurations, à savoir :

1. **Terminaison** des reconfigurations.
2. **Atomicité** des reconfigurations.
3. **Isolation** entre plusieurs reconfigurations.
4. **Consistance des états**, en particulier de l'état final de l'application, et de la transition.

En dehors de la terminaison, qui est garantie par la conception même du langage par l'intermédiaire des structures de contrôle disponibles, nous avons choisi de déléguer la gestion de ces critères à l'implémentation. Ce choix nous donne une plus grande liberté quant aux stratégies d'implémentation sans compromettre la sûreté des reconfigurations.

Notons que les critères de consistance énoncés plus haut n'ont de sens qu'en ce qui concerne les reconfigurations de plus haut niveau (*top-level*), c'est-à-dire initiées par du code Java par l'intermédiaire de la méthode `FScriptInterpreter.execute()`. L'intégrité structurelle en particulier peut ne pas être toujours vérifiée en cours de reconfiguration, tant que le système garantit qu'elle l'est avant et après la reconfiguration. Cette similitude avec les transactions des bases de données – que nous avions déjà évoqué plus haut – nous a conduit à concevoir le modèle d'exécution de haut niveau des reconfigurations FScript comme des transactions. Ainsi, l'algorithme principal qui contrôle l'exécution des reconfigurations et qui constitue le schéma de fonctionnement de la méthode `FScriptInterpreter.execute()` est le suivant :

**T1.** Démarrage de la reconfiguration.

**T2.** Exécution de l'instruction FScript. Si une erreur fatale se produit (en pratique une exception est levée), annulation de la reconfiguration et retour à l'état initial.

**T3.** Test de validité du nouvel état du système.

**T4.** Si le nouvel état est consistant, validation de la transaction / reconfiguration.

**T5.** Sinon, annulation et retour à l'état initial.

Ce schéma de fonctionnement garantit l'intégrité structurelle de l'application Fractal reconfigurée (vérifiée à l'étape **T3**) et l'atomicité de la reconfiguration (par le retour à l'état initial en cas d'erreur). Cependant, il ne s'agit que d'un schéma, qui peut donner lieu à plusieurs implémentations. Afin de permettre d'expérimenter plusieurs approches, l'implémentation de l'interprète FScript est conçue selon le schéma *Pont* (*Bridge*) [Gamma et al., 1994]. L'interprète est séparé en deux parties : la première, fixe, est chargée de l'interprétation des constructions du langage (flot de contrôle, gestion de variables, etc.) ; la seconde est représentée par l'interface `FractalModel` et est une *abstraction* de l'application sur laquelle les reconfigurations s'appliquent. Ainsi, l'interprète FScript ne manipule jamais directement les interfaces de programmation Fractal, mais passe toujours par l'intermédiaire d'une implémentation de `FractalModel`. Cette interface Java a la signature suivante :

```
public interface FractalModel {
    FractalValueFactory getValueFactory();
    void loadContributedFunctions(Environment env);
    boolean startReconfiguration(FScriptInterpreter inter, Statement stat);
    boolean isValidReconfiguration();
    void commitReconfiguration();
    void cancelReconfiguration();
}
```

La première méthode, `getValueFactory()` renvoie une fabrique de valeurs FScript qui permet une implémentation spécifique des types de données FScript, et en particulier des différents types de nœuds. La deuxième méthode est utilisée par l'interprète au démarrage pour charger toutes les fonctions et actions primitives supportées par ce modèle. Cet ensemble doit contenir au moins toutes les procédures

standards définies dans ce chapitre et dans l'annexe A. Enfin, les quatre autres méthodes sont utilisées par l'interprète pendant la reconfiguration et correspondent respectivement aux étapes **T1**, **T3**, **T4** et **T5** du schéma d'exécution décrit plus haut.

Cette conception permet de déléguer à l'abstraction `FractalModel` tous les aspects de la reconfiguration qui interagissent directement avec Fractal. Il existe de nombreuses stratégies d'implémentation possibles, qui se distinguent par le moment où elles détectent les erreurs et la façon dont elles les traitent. Ainsi, si l'on dispose d'analyses statiques poussées permettant de détecter à l'avance certaines erreurs, il est possible de les effectuer dans la méthode `startReconfiguration()`, à laquelle toutes les informations nécessaires sur la reconfiguration sont passées en paramètre. Si ces analyses détectent que la reconfiguration est invalide, la méthode renvoie *faux* et l'interprète annule la reconfiguration et considère qu'elle a échouée.

Cependant, nous ne disposons pas actuellement de telles analyses statiques, et nous avons donc choisi une implémentation de `FractalModel` basée sur une stratégie différente que l'on pourrait appeler « *try / repair* » : la reconfiguration est appliquée « en direct » à l'application, mais le modèle garde un journal des actions effectuées contenant toutes les informations nécessaires pour pouvoir les annuler. Si à la fin de la reconfiguration le nouvel état atteint n'est pas valide, ce journal est utilisé pour défaire (*undo*) la reconfiguration et revenir à l'état initial. Plus concrètement :

**T1.** Aucune analyse statique n'est effectuée, et la méthode `startReconfiguration()` se contente donc d'initialiser quelques structures de données.

**T2.** Toutes les procédures primitives demandées par l'interprète sont appliquées immédiatement et directement sur les composants Fractal constituant l'application, mais lorsqu'une *action* est appliquée et modifie l'application, une action primitive inverse, qui annule la première (*undo*) est ajoutée dans un *journal* correspondant à cette transaction de reconfiguration.

De plus, à chaque fois qu'une action primitive stoppe implicitement un composant afin de pouvoir être appliquée sans erreur, ce composant est ajoutée dans un ensemble $S$. Si jamais une opération `stop()` est explicitement invoquée sur l'un de ces composants pendant la reconfiguration, il est retiré de l'ensemble $S$.

Si jamais une erreur irrécupérable se produit, c'est-à-dire si une exception est levée lors de l'exécution d'une action primitive, cette exception est propagée vers l'interprète pour qu'il déclenche l'annulation de la reconfiguration.

**T3.** La méthode `isValidReconfiguration()` effectue les tests suivants :

1. Pour tous les composants de l'ensemble $S$ et pour tous leurs sous-composants directs et indirects (qui ont aussi été stoppés puisque cette opération est récursive dans l'implémentation que nous utilisons), on vérifie que toutes les interfaces clientes obligatoires sont bien connectées. Cela se fait très simplement en évaluant l'expression FPath suivante :

   ```
   descendant-or-self::*[interface::*[client()][mandatory()][not(bound())]]
   ```

   qui doit renvoyer un ensemble vide de nœuds. Si ce n'est pas le cas, cela signifie que ces composants ne peuvent pas être redémarrés sans erreur.

2. Pour chaque composant ayant été impliqué dans la reconfiguration, toutes ces contraintes métiers définies dans son `constraints-controller` sont testées (cf. 4.3.2).

Si l'un de ces deux prédicats n'est pas vérifié, alors le nouvel état final est considéré comme inconsistant.

Ces tests sont suffisants pour garantir l'intégrité structurelle de l'état final puisque, arrivé à cette étape de la reconfiguration, toutes les primitives étant appliquées au fur et à mesure, les autres critères ont été vérifiés pendant l'exécution.

**T4.** La validation de la reconfiguration (`commitReconfiguration()`) consiste simplement à redémarrer tous les composants de l'ensemble $S$. Les tests effectués à l'étape précédente nous garantissent qu'aucune erreur ne peut se produire ici.

**T5.** Enfin, l'annulation de la reconfiguration se fait en rejouant à l'envers le journal des actions effectuées, ou plus précisément de leurs inverses (*undo*). Étant donné l'implémentation de Fractal que nous utilisons, toutes les actions primitives décrites dans ce chapitre ont une action inverse bien définie qui est garantie de s'exécuter sans erreur si l'état de départ du système avant l'annulation correspond à l'état final après l'exécution initiale. Cette implémentation nous garantit aussi que les opérations primitives elles-mêmes sont atomiques, c'est-à-dire que si une exception est levée pendant l'exécution d'une action primitive, l'état du système n'a pas été modifié.

Ainsi, si l'on considère que la reconfiguration globale est une fonction $R$ constituée d'une série de fonctions primitives $R = f_1 \circ \cdots \circ f_n$, alors l'annulation de la reconfiguration consiste à appliquer la fonction inverse $R^{-1} = f_n^{-1} \circ \cdots \circ f_1^{-1}$ constituée des inverses des fonctions primitives, appliquées à l'envers.

Cette stratégie d'implémentation « optimiste » nous garantit dont bien les propriétés d'*intégrité structurelle* et d'*atomicité*.

L'*intégrité comportementale*, qui indique qu'aucun message n'est perdu pendant l'exécution de la reconfiguration nous est garantie automatiquement par l'implémentation de Fractal que nous utilisons. En effet, dans cette implémentation lorsqu'un composant est dans un état `STOPPED`, tous les messages envoyés vers l'une de ses interfaces de services sont bloqués pour être rejoués – dans le même ordre – lorsque le composant est redémarré. Puisque toutes les opérations primitives qui modifient les canaux de communications par lesquels passent les messages stoppent le composant impliqué avant de le modifier, aucune message ne peut donc être perdu.

Le dernier critère à vérifier est celui de l'*isolation* entre plusieurs reconfigurations : les étapes intermédiaires d'une reconfiguration $R_1$ ne doivent pas être visibles de la reconfiguration $R_2$ car même si $R_1$ est consistante prise dans sa globalité, certains de ces états intermédiaires peuvent ne pas l'être[7]. Ce critère, qui est lié à la sérialisabilité des reconfigurations, est très complexe à implémenter de façon efficace, surtout étant donné notre stratégie d'implémentation actuelle qui modifie réellement les composants au fur et à mesure. Notre implémentation actuelle, potentiellement très inefficace mais correcte, consiste à ne permettre qu'à une seule reconfiguration à la fois de s'exécuter. Si une reconfiguration est initiée pendant qu'une autre est encore en cours d'exécution, elle sera mise en attente jusqu'à la fin de la première. Nous avons étudié quelques approches qui permettraient une prise en compte plus efficace de ce critère mais ces études n'ont pas encore abouties faute de temps.

Pour conclure cette section, nos disposons donc d'une implémentation concrète du langage FScript, basée sur une stratégie optimiste de type « *try / repair* », qui garantit tous les critères de consistance que nous avions identifiés. De plus, la conception de l'interprète permet d'expérimenter facilement d'autres stratégies d'implémentation.

## 6.6 Conclusion : limitations et extensions futures

Dans ce chapitre, nous avons décrit la conception et l'implémentation de FScript, un langage dédié à la spécification et l'exécution des reconfigurations architecturales dynamiques et consistantes d'applications Fractal. Les propriétés combinées du langage et de son implémentation offrent un certain nombre de garanties importantes concernant la consistance des reconfigurations appliquées, qui permettent d'utiliser FScript pour modifier des applications sans craindre de les rendre inutilisables.

FPath, un sous-ensemble de FScript purement sans effet de bords a été décrit séparément, car aussi bien au niveau de sa conception que de son implémentation, il a été créé pour pouvoir être utilisé seul. Ce sous-langage permet, entre autres, de naviguer dans des architectures Fractal et de sélectionner des composants, interfaces ou attributs qui vérifient certaines propriétés.

L'implémentation actuelle de FScript souffre de certaines limitations. Cependant, elle a été conçue spécifiquement pour pouvoir évoluer sans modifier la sémantique du langage lui-même.

[7] En pratique, état donnée notre implémentation actuelle, seules les contraintes métiers définies grâce à l'interface `contraints-controller` peuvent être violées au cours d'une reconfiguration.

Une des améliorations possibles serait l'ajout d'analyses, soit statiques sur le code d'une reconfiguration, soit semi-statiques au début d'une transaction de reconfiguration (au moment ou le ou les composants impliqués sont connus). De telles analyses permettraient de détecter à l'avance certaines erreurs que l'implémentation actuelle ne détecte qu'après coup. La modélisation sous-jacente à FPath, qui représente un ensemble de composants sous la forme d'un graphe, pourra sans doute servir de base à la formalisation de FScript nécessaire à la création de telles analyses.

Une autre limitation de l'implémentation actuelle est l'impossibilité d'exécuter plusieurs reconfigurations en parallèle. Or, dans de nombreux cas pratiques de telles reconfigurations ne se chevauchent pas, c'est-à-dire que les ensembles de composants impliqués sont disjoints. Malheureusement, il est difficile, voire peut-être impossible, de prévoir à l'avance si deux reconfigurations risquent de se chevaucher. Le verrou global utilisé actuellement est un mécanisme à grain beaucoup trop gros, et des mécanismes de verrou plus fin[8] devraient permettre l'exécution de reconfigurations parallèles tout en conservant toutes les propriétés de consistance, et en particulier l'isolation.

[8]Par exemple un verrou par composant, mais une implémentation naïve risquerait d'introduire des interbloquages.

# Chapitre 7

# Politiques d'adaptation SAFRAN

## Sommaire



Ce chapitre décrit un langage dédié pour la spécification de politiques d'adaptation dynamique. Ce langage constitue le cœur du système SAFRAN : il intègre en un tout cohérent l'ensemble des contributions décrites dans les chapitres précédents afin de permettre la spécification et l'intégration dynamique de l'*aspect d'adaptation* dans les applications à base de composants Fractal.

## 7.1 Introduction

L'objectif principal de nos travaux est de rendre plus simple la création d'applications adaptatives. Pour cela, notre approche repose sur les choix suivants :

1. considérer l'adaptation dynamique comme un *aspect*, les politiques d'adaptation étant modularisées en dehors du code métier de l'application afin de pouvoir y être intégrées dynamiquement ;
2. utiliser la programmation par composants, et plus spécifiquement le modèle Fractal, pour la construction d'applications *adaptables* sur lesquelles les politiques d'adaptation peuvent se greffer ;
3. utiliser un langage dédié pour la spécification des politiques d'adaptation ;
4. utiliser le paradigme des règles réactives ECA pour structurer ces politiques.

Nous avons déjà décrit l'architecture générale du système SAFRAN qui implémente cette approche dans le chapitre 4. Cette architecture a été conçue de façon modulaire, chacun de ses éléments pouvant être réutilisé dans d'autres applications. La figure 7.1 représente les différents éléments qui constituent SAFRAN.

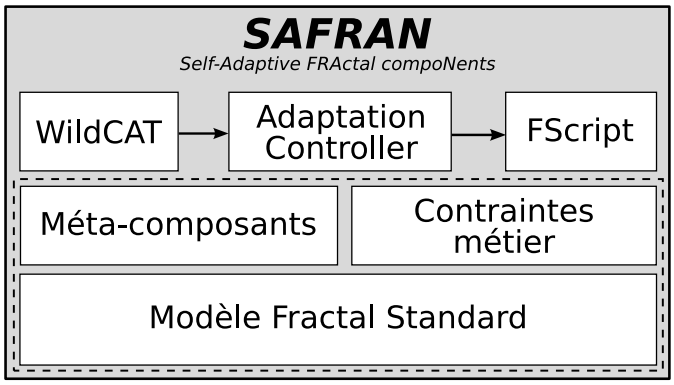


FIG. 7.1 – Les différents éléments constituant le système SAFRAN.

Le modèle Fractal et les deux extensions que nous y avons ajoutées ont été présentées en détail dans les sections 4.2, 4.3.1 et 4.3.2. Le système d'observation et de détection des changements du contexte d'exécution WildCAT a été décrit dans le chapitre 5, et le langage dédié FScript permettant de programmer des reconfigurations dynamiques consistantes d'applications Fractal est quant à lui présenté dans le chapitre 6. Enfin, nous avons aussi décrit l'interface abstraite `adaptation-controller` (Section 4.4.1) conçue pour permettre d'attacher dynamiquement des politiques d'adaptation à des composants Fractal afin de les rendre adaptatifs :

```
interface AdaptationController {
  void attachFcPolicy(any policy) throws InvalidPolicyException;
  void detachFcPolicy(any policy) throws NoSuchPolicyException;
  any[] getFcPolicies();
}
```

Cette définition abstraite n'indique pas le type concret des politiques utilisées, ni la façon de les définir, et encore moins leur mode d'exécution.

Ce chapitre conclut donc notre présentation de SAFRAN en décrivant l'implémentation concrète de cette extension `adaptation-controller`, sous la forme d'un langage dédié permettant de définir des *politiques d'adaptation*. Le rôle de ces politiques est d'implémenter les stratégies d'adaptation à appliquer aux composants d'une application afin de les rendre *adaptatifs*. Une fois définies grâce au langage SAFRAN[1] décrit dans ce chapitre, ces politiques pourront alors être utilisées avec le contrôleur Fractal ci-dessus, dont l'implémentation se chargera de leur exécution.

Étant donnée la nature réactive du processus d'adaptation (cf. Section 4.4.2), nous avons choisi de structurer les politiques d'adaptation SAFRAN sous la forme de règles réactives inspirées du paradigme ECA issu du domaine des bases de données actives [Dittrich et al., 1995; Collet, 1996]. Une politique d'adaptation SAFRAN est donc constituée d'un ensemble de règles réactives, chacune constituée de trois éléments :

1. la spécification du type d'événement (éventuellement composite) qui active la règle lorsqu'il se produit ;
2. une condition (expression booléenne) servant de garde ;

[1] SAFRAN est à la fois le nom du langage dédié et du système dans son ensemble.

3. une action de reconfiguration, éventuellement composée d'une séquence de plusieurs actions primitives parmi celles disponibles.

Les différentes contributions présentées dans les chapitres précédents sont destinées à implémenter les différents éléments des règles réactives :

- Le système WildCAT (Chapitre 5) est utilisé pour détecter les *événements* exogènes (venant de l'extérieur de l'application) qui représentent les évolutions du contexte d'exécution.
- Le langage FPath (Section 6.3) est utilisé pour exprimer les *conditions* qui servent de garde aux règles réactives.
- Enfin, la langage FScript (Section 6.4) est utilisé pour exprimer les *actions* de reconfiguration et pour les exécuter de façon sûre.

Ainsi, toutes les contributions décrites jusqu'à présent trouvent leur place dans SAFRAN, chacune servant à implémenter une partie du langage dédié aux politiques d'adaptation.

Dans ce chapitre, nous présentons donc ce langage dans sa globalité, en nous concentrant sur les aspects qui n'ont pas encore été décrits. Dans un premier temps, nous décrivons comment spécifier la partie $E$ d'une règle réactive, qui indique le ou les événements qui doivent la déclencher (Section 7.2). Les deux autres parties des règles ont déjà été décrites individuellement : FPath pour la condition $C$ et FScript pour l'action $A$. Le reste du chapitre présente donc les politiques d'adaptation dans leur globalité, en commençant par les aspects syntaxiques et structuraux (Section 7.3) avant de présenter leur modèle d'exécution (Section 7.4).

## 7.2 Spécification et détection d'événements

### 7.2.1 Introduction

Dans cette section, nous décrivons la syntaxe et la sémantique des *descripteurs d'événements* qui constituent la première partie des règles réactives ECA. Le rôle de ces descripteurs est de spécifier les circonstances qui doivent déclencher une règle, et donc – éventuellement – une adaptation. En effet, nous avons déjà vu que le processus d'adaptation est par nature réactif : l'adaptation d'un système ne devient nécessaire que si une modification « significative » se produit, soit dans le contexte d'exécution, soit dans l'application elle-même. L'objectif des descripteurs d'événements est donc de permettre à la fois la *spécification* des événements significatifs dans le cadre des règles ECA, puis la *détection* de l'occurrence des événements en question.

Cette problématique de la spécification et détection d'événements se retrouve dans de très nombreux domaines d'applications, comme par exemple la détection d'intrusions, la gestion d'applications distribuées, et bien sûr les bases de données actives. Il s'agit d'un problème complexe qui a donné lieu à des travaux spécifiques, allant des calculs formels comme [Shanahan, 1999; Courtenage, 2002] aux architectures logicielles spécifiques [Marzullo et al., 1991; Schade, 1997; Moreto and Endler, 2001], en passant par les langages dédiés comme SNOOP [Chakravarthy and Mishra, 1994] et GEM [Mansouri-Samani and Sloman, 1997].

Dans le cadre de SAFRAN, nous nous sommes inspirés de certains de ces travaux afin de développer à la fois un langage dédié et une architecture logicielle associée qui permettent la spécification et la détection des événements qui peuvent déclencher une adaptation, tout en s'intégrant parfaitement dans la structure et le mode d'exécution des politiques d'adaptation.

Les objectifs qui nous ont guidé pour la conception et l'implémentation de ce langage sont les suivants :

1. Le *pouvoir d'expression* du langage doit être suffisant pour pouvoir exprimer toutes les circonstances qui peuvent demander une réaction de la part d'une politique SAFRAN afin d'adapter l'application.
2. Le langage doit rester suffisamment simple pour être *lisible* et *compréhensible* par le programmeur des politiques d'adaptation.

Avant de décrire les événements eux-mêmes, précisons quelques notions spécifiques à ce domaine.

**Descripteur d'événement *vs* occurrence d'événement.** On distingue les *descripteurs d'événements* des *occurrences d'événements*. Les descripteurs d'événements décrivent une ou plusieurs situations que le système doit détecter. Les occurrences représentent les événements qui se produisent réellement, et peuvent être vus comme des instances des descripteurs.

**Événements primitifs *vs* événements composites.** Parmi les descripteurs d'événements, on distingue les descripteurs d'événements primitifs des descripteurs composites. Le système WildCAT présenté dans le chapitre 5 peut offrir des informations très complètes et précises concernant l'environnement d'une application, mais ces informations ne correspondent qu'à un instantané de l'état de l'environnement à un moment donné. Pour aller plus loin, nous avons besoin de raisonner sur les évolutions de cet environnement au cours du temps. Nous introduisons pour cela la possibilité de spécifier et détecter des événements composites [Courtenage, 2002], décrivant l'enchaînement d'événements primitifs au cours du temps. À partir des événements primitifs, des opérateurs de composition (séquence, alternative, conjonction, ...) permettent d'exprimer des conditions complexes concernant l'évolution de l'environnement de l'application.

**Événements endogènes *vs* événements exogènes.** En plus de devoir réagir aux évolutions de son environnement, une application peut aussi être amenée à s'adapter en réaction à des changements internes, comme par exemple la création d'un nouveau composant, l'invocation d'une certaine méthode, etc. Ce second cas est traité de façon homogène par notre système, en introduisant un nouveau type d'événements primitifs, dits *endogènes* (par opposition aux événement *exogènes* venant de l'extérieur), correspondant aux événements se produisant à l'intérieur même de l'application (reconfigurations, envoi et réception de messages...). Bien entendu, il est possible lors de la spécification d'un événement composite, de mélanger des événements endogènes et exogènes.

**Correspondance entre le temps d'occurrence des événements et le temps de la détection.** Une caractéristique importante de la détection d'événement est qu'elle doit s'effectuer en « temp réel ». Bien que la détection d'événements soit similaire à la recherche de motif (*pattern matching*, expressions régulières...), le temps de la détection doit correspondre au temps d'occurrence ; il ne s'agit pas d'analyser une trace d'événements pour y détecter *a posteriori* certains motifs, mais de les détecter au moment où ils se produisent. Cette distinction a des conséquences importantes sur les opérateurs de composition utilisables : le retour en arrière est impossible, le futur n'est pas borné, et la détection d'un événement ne doit pas dépendre des événements ultérieurs. De plus, les performances doivent être une priorité majeure, même en regard du pouvoir d'expression et de la complexité du langage.

**Génération des événements : modèle logique *vs* modèle opérationnel.** Du point de vue logique, tous les événements primitifs qui se produisent lors de l'exécution de l'application sont effectivement « générés » (par WildCAT ou par les composants Fractal concernés) et notifiés à tous les détecteurs présents dans le système. En pratique, une telle implémentation naïve serait beaucoup trop inefficace, puisque seul un petit pourcentage des occurrences d'événements sont significatifs, au sens où ils peuvent déclencher des adaptations. L'implémentation est donc libre de ne générer que certaines des occurrences d'événements, tant qu'elle garantit que le comportement observable du système du point de vue de la détection des événements n'est pas modifié[2]. Concrètement, cela signifie par exemple que tant qu'aucune politique d'adaptation ne réagit à la quantité de mémoire disponible sur le système, les événements correspondants à ce paramètre du contexte ne sont pas générés, et la sonde WildCAT correspondante n'est pas déployée.

**Capture de données lors de la détection.** La spécification des événements décrite dans cette section est destinée à faire partie des règles réactives constituant les politiques d'adaptation SAFRAN. Afin de permettre une meilleure intégration des différents éléments des règles ECA, le résultat de la détection

[2]Sauf bien entendu en terme de performances.

d'un événement n'est pas un simple signal « l'événement s'est produit ». En plus de cette information, la notification d'un événement peut capturer certaines des occurrences qui ont participé à sa détection afin de rendre ces informations disponibles à la suite du processus d'adaptation. Ce mécanisme est similaire à la capture des sous-chaînes dans les expressions régulières (`$1`, `$2`, etc. correspondants aux groupes de parenthèses de l'expression).

### 7.2.2 Événements primitifs

#### Introduction

Nous décrivons dans cette section les différents types d'événements primitifs qui peuvent être détectés par les politiques d'adaptation SAFRAN. Lorsqu'un événement quel qu'il soit se produit, son *occurrence* est *réifiée*[3] sous la forme d'un objet structuré (enregistrement) dont les différents champs décrivent l'occurrence en question. Chaque type d'événement peut définir des champs spécifiques, mais les suivants sont communs à tous :

1. `type` : une chaîne de caractère qui indique à quel type de phénomène l'événement correspond, comme par exemple l'invocation d'une méthode (`"message-received"`) ou la modification d'un paramètre du contexte (`"changed"`).
2. `source` : un objet Java qui indique l'origine de l'événement. Suivant les cas, il peut s'agir d'un composant de l'application ou d'un élément du contexte (ressource ou attribut WildCAT).
3. `timestamp` : un nombre entier qui représente l'instant précis où l'événement s'est produit (tel qu'indiqué par la méthode Java `System.currentTimeMillis()`).

Pour chaque type d'événement primitif supporté, il existe un *descripteur* correspondant qui permet de détecter les occurrences de ce type. La forme générale de ces descripteurs est :

```
type_événement(arguments | predicat)
```

- `type_événement` correspond au type d'événement à détecter (champ `type`).
- `arguments` est une liste de paramètres qui permettent de spécialiser le descripteur pour ne détecter que certaines occurrences du type `type`. La forme exacte de ces paramètres dépend du type d'événement.
- `predicat` est une expression booléenne qui peut être omise (dans ce cas, la barre verticale doit aussi être omise) et qui permet de filtrer plus finement les occurrences détectées par le descripteur en fonction des valeurs des champs de l'occurrence. À l'intérieur de l'expression, la valeur de chaque champ est accessible par l'intermédiaire d'une variable au nom correspondant (par exemple `$timestamp`).

Ainsi, pour réagir à une modification quelconque de la quantité de mémoire disponible, il suffit d'écrire :

```
changed(sys://storage/memory#free)
```

Ici, `changed` est un type d'événement (décrit plus loin) qui permet de détecter des changements dans le contexte d'exécution, tel qu'il est représenté par WildCAT. `sys://storage/memory#free` est l'expression WildCAT (ici un simple chemin désignant un attribut) dont la valeur doit être observée.

Si l'on n'est intéressé que par les *diminutions* de cet attribut plutôt que par tous les changements, l'expression appropriée est alors

```
changed(sys://storage/memory#free | $old-value < $new-value)
```

Dans ce cas, le test est une expression qui utilise les champs `old-value` et `new-value` définis dans les occurrences du type `changed`.

[3] Pour des raisons de performances, une implémentation concrète peut décider de ne pas réifier certaines occurrences tant que cela ne modifie pas le comportement du système.

Nous décrivons maintenant l'ensemble des événements primitifs supportés par SAFRAN, en indiquant à chaque fois les phénomènes auxquelles ils correspondent, les différents champs qui décrivent les occurrences, et la syntaxe des descripteurs primitifs correspondant. Tous les événements sont détectés *a posteriori*, c'est-à-dire que le phénomène qu'ils décrivent s'est déjà produit au moment de la détection. Les différents types d'événements supportés sont regroupés dans deux catégories suivant leur origine : d'une part les événements *endogènes* qui correspondent à (certains aspects) de l'exécution de l'application Fractal elle-même, et d'autre part les événements *exogènes*, qui correspondent aux modifications du contexte d'exécution (tel que représenté par WildCAT). Au-delà de leur origine, ces deux catégories diffèrent par leur mode de détection : les événements endogènes sont détectés de façon synchrone (bloquante) par rapport à leur occurrence dans l'application, alors que les événements exogènes sont par nature asynchrones relativement à l'exécution du programme.

### Événements primitifs endogènes

Les événements primitifs endogènes permettent à une politique d'adaptation de réagir à l'exécution de l'application adaptée elle-même : d'une part le *flot d'exécution*, et d'autre part les modifications de sa *structure*.

Dans le cadre de Fractal, le flot d'exécution se limite à l'envoi et à la réception de messages, représentés les trois types d'événements suivants, que l'on peut rapprocher des *points de jonction* d'un langage d'aspects (en particulier dans le cas d'EAOP [Douence et al., 2002] qui traite explicitement ces points de jonction comme des événements). Ces trois types d'événements sont :

- `message-received`, qui est déclenché lors de la réception d'un message par une interface de service d'un composant ;
- `message-returned`, qui indique le retour avec succès d'une telle invocation ;
- et `message-failed`, qui indique au contraire qu'une invocation a générée une erreur.

Ces trois types d'événements partagent la même syntaxe de descripteur, qui permet de sélectionner les interfaces et les méthodes que l'on veut observer. Ainsi, `message-received($c/interface::logger)` détecte les invocations de n'importe quelle méthode de l'interface `logger` de `$c`, alors que l'expression `message-failed($c/interface::*)` détecte les erreurs survenues sur n'importe quelle interface de service de ce même composant.

| **Réception de message** | |
|---|---|
| **Syntaxe** | `message-received(interfaces)`<br>`message-received(interfaces, method)` |
| **Description** | Indique la réception d'un message (i.e. l'invocation d'une méthode) sur une interface de service d'un composant. `interfaces` désigne l'ensemble des interfaces de composants que le descripteur doit observer. `method` est le nom optionnel d'une méthode spécifique ; s'il est omis, toutes les méthodes des interfaces désignées sont considérées. |
| **Champs** | `source` : l'interface qui a reçu le message<br>`method` : la méthode invoquée<br>`args` : les paramètres de l'invocation (`Object[]`) |

| **Retour d'une invocation avec succès** | |
|---|---|
| **Syntaxe** | `message-returned(interfaces)`<br>`message-returned(interfaces, method)` |
| **Description** | Indique que l'exécution d'un message s'est terminée normalement, renvoyant éventuellement une valeur de retour. La syntaxe et la sémantique du descripteur est la même que pour `message-received`. |
| **Champs** | `source` : l'interface qui a reçu le message<br>`method` : la méthode invoquée<br>`args` : les paramètres de l'invocation (`Object[]`).<br>`return-value` : la valeur de retour de l'invocation |

| Retour d'une invocation avec erreur | |
|---|---|
| **Syntaxe** | `message-failed(interfaces)`<br>`message-failed(interfaces, method)` |
| **Description** | Indique que l'exécution d'un message s'est terminée anormalement par la levée d'une exception. La syntaxe et la sémantique du descripteur est la même que pour les deux types précédents. |
| **Champs** | `source` : l'interface qui a reçu le message<br>`method` : la méthode invoquée<br>`args` : les paramètres de l'invocation<br>`exception` : l'exception qui a été levée |

Les reconfigurations structurelles correspondent aux méthodes disponibles sur les interfaces de contrôle des composants Fractal. Bien qu'en pratique ces reconfigurations soient déclenchées *in fine* par des invocations de méthodes Java, celles-ci génèrent des types d'événements spécifiques plutôt que les événements génériques `message-*`, qui sont réservés pour les interfaces de service. Les types d'événements correspondants aux reconfigurations sont les suivants.

| Création d'un composant | |
|---|---|
| **Syntaxe** | `component-created()` |
| **Description** | Indique la création d'un nouveau composant. Le descripteur ne prend pas de paramètres. |
| **Champs** | `source` : le composant nouvellement créé |

| Cycle de vie des composants | |
|---|---|
| **Syntaxe** | `component-started(components)`<br>`component-stopped(components)` |
| **Description** | Indiquent le démarrage ou l'arrêt d'un composant, correspondant respectivement à l'invocation des méthode `startFc()` et `stopFc()` du contrôleur de cycle de vie (`LifeCycleController`). La paramètre `components` désigne les composants observés. Par exemple, si `$c` désigne un composite, l'expression `component-started($c/child::*)` permet de détecter le démarrage de n'importe quel sous-composant direct de `$c`. |
| **Champs** | `source` : le composant démarré ou stoppé |

| Paramétrage d'un composant | |
|---|---|
| **Syntaxe** | `parameter-changed(parameters)` |
| **Description** | Indique qu'un paramètre de configuration d'un composant a été modifié (par l'interface `attribute-controller`). `parameters` désigne le ou les paramètres à observer, par exemple `$c/@*` ou plus spécifiquement `$c/attribute::size`. |
| **Champs** | `source` : le paramètre qui a été modifié<br>`old-value` : l'ancienne valeur du paramètre<br>`new-value` : sa nouvelle valeur |

| Modification du contenu des composites | |
|---|---|
| **Syntaxe** | `subcomponent-added(composites)`<br>`subcomponent-removed(composites)` |
| **Description** | Indiquent respectivement l'ajout ou le retrait d'un sous-composant dans un composite. Dans les deux cas `composites` est l'ensemble des composants composites dont le contenu doit être observé. |
| **Champs** | `source` : le composite dont le contenu a été modifié<br>`sub-component` : le sous-composant ajouté ou retiré |

| Modification des connexions | |
|---|---|
| **Syntaxe** | `binding-created(client-interfaces)`<br>`binding-destroyed(client-interfaces)` |
| **Description** | Indiquent respectivement la création ou la destruction d'une connexion entre deux interfaces de composants. Dans les deux cas, le paramètre `client-interfaces` désigne la liste des interfaces clientes dont les connexions doivent être observées. |
| **Champs** | `source` : l'interface cliente qui a été connectée ou déconnectée<br>`server-interface` : l'interface serveur à laquelle `source` a été connectée ou déconnectée |

| Modification du lien méta | |
|---|---|
| **Syntaxe** | `metalink-changed(components)` |
| **Description** | Ce type d'événement indique que le lien méta qui relie un composant réflexif « de base » à son méta-composant a été modifié. La paramètre `components` désigne les composants réflexifs à observer. |
| **Champs** | – `source` : le composant de base dont le lien méta a été modifié ;<br>– `old-meta` : l'ancien méta-composant ;<br>– `new-meta` : le nouveau méta-composant. |

| Manipulation des politiques d'adaptation | |
|---|---|
| **Syntaxe** | `policy-attached(components)`<br>`policy-detached(components)` |
| **Description** | Indiquent respectivement qu'une politique d'adaptation a été attachée ou détachée d'un composant adaptatif SAFRAN. Le paramètre `components` désigne les composants adaptatifs à observer. |
| **Champs** | `source` est le composant adaptatif auquel on a attaché ou détaché une politique d'adaptation<br>`policy` est la politique d'adaptation qui a été attachée ou détachée |

### Événements primitifs exogènes

Les événements primitifs exogènes correspondent à des modifications dans le contexte d'exécution tel que modélisé par WildCAT (cf. Chapitre 5). Les différents types d'événements supportés sont les suivants.

| Changement de valeur d'une expression | |
|---|---|
| **Syntaxe** | `changed(expr)` |
| **Description** | Indique un changement quelconque de la valeur d'une expression `expr`. Cette dernière peut être n'importe quelle expression qui pourrait être utilisée pour définir un attribut synthétique dans WildCAT (cf. Section 5.5.3, page 87). |
| **Champs** | `source` : l'élément du contexte (ressource ou attribut) qui est à l'origine du changement de valeur de l'expression<br>`old-value` : la valeur précédente de l'expression<br>`new-value` : sa nouvelle valeur |

| Réalisation d'une condition | |
|---|---|
| **Syntaxe** | `realized(expr)` |
| **Description** | Indique le passage à *vrai* d'une expression booléenne WildCAT. Ce type d'événement est similaire au précédent, mais valable uniquement pour des expressions booléennes, et émis seulement lors d'une transition de *faux* vers *vrai*. Il s'agit donc d'un raccourci syntaxique pour `changed(expr \| $new-value and not($old-value))`. |
| **Champs** | `source` : l'élément du contexte (ressource ou attribut) qui est à l'origine du changement de valeur de l'expression |

| Modifications de la structure du contexte | |
|---|---|
| **Syntaxe** | `appears(path)`<br>`disappears(path)` |
| **Description** | Indiquent respectivement l'apparition ou la disparition d'un attribut ou d'une ressource WildCAT sur un chemin donné. Le dernier élément du chemin peut être un caractère *joker*, comme par exemple dans `appears(sys ://devices/usb/*)`. |
| **Champs** | `source` : l'élément (ressource ou attribut) apparu ou disparu. |

Notre système ne définit pas de type d'événement temporels absolus spécifique, permettant de déclencher une règle à un moment déterminé, comme par exemple tous les soirs de semaine à trois heures du matin. Cependant, les descripteurs d'événements exogènes décrits ci-dessus peuvent remplir ce rôle si WildCAT réifie le temps absolu (date et heure) d'une façon appropriée. Il devient alors possible d'écrire des descripteurs d'événements tels que

```
realized(not(weekend(clock://date#day_of_week))
         and clock://time#hour == 3
         and clock://time#minute == 0)
```

### 7.2.3 Descripteurs d'événements composites

Les descripteurs vus jusqu'à présent ne permettent de détecter que des occurrences d'événements ponctuelles. Nous introduisons maintenant de nouveaux opérateurs qui permettent de modifier un descripteur d'événement ou d'en combiner plusieurs. Ces opérateurs augmentent le pouvoir d'expression de notre système de détection, en particulier en lui permettant de réagir aux *évolutions* du système plutôt qu'à son état instantané.

Les deux premiers opérateurs sont des opérateurs *logiques*, qui combinent au moins deux descripteurs :

$evt_1$ `or` $evt_2$ **(`or ...`) :** se déclenche dès que l'un quelconque des événements mentionnés est détecté.

$evt_1$ `and` $evt_2$ **(`and ...`) :** se déclenche dès que tous les événements mentionnés ont été détectés, quel que soit l'ordre.

Les trois autres opérateurs sont de type *chroniques*, c'est-à-dire qu'ils sont sensibles au moment où les événements se produisent (dans l'absolu ou relativement les uns aux autres) :

$evt_1$ **`,`** $evt_2$ **:** Permet d'exprimer la *séquence* de deux (ou plus) événements. L'événement composite se déclenche lorsqu'$evt_2$ est détecté, si et seulement si $evt_1$ a été détecté auparavant.

**`after(evt, delay)` :** Permet d'introduire un délai dans la détection d'un événement. Cet événement composite se déclenche `delay` secondes après la détection d'`evt`.

**`never(begin, evt, end)` :** Permet d'exprimer une certaine forme de négation, c'est-à-dire la *non-occurrence* d'un événement. Par définition, la non-occurrence d'un événement n'est pas détectable en général, puisque le temps est infini aussi bien vers le passé que vers le futur. Pour pouvoir détecter l'absence d'un événement, nous devons donc borner l'intervalle de temps dans lequel l'absence de l'événement doit être détecté. Pour cela, l'opérateur `never` prend en paramètre, en plus de l'événement `evt` dont la non-occurrence doit être détectée, deux descripteurs, `begin` et `end` dont la détection marque respectivement le début et la fin de l'intervalle de temps considéré. Ainsi, `never(begin, evt, end)` détecte la séquence `begin, end`, si et seulement si `evt` n'est pas détecté entre les deux.

**`never(begin, evt, delay)` :** Forme spéciale de `never()` qui utilise un délai, relatif à l'occurrence de `begin`, pour indiquer la fin de l'intervalle de détection. `never(begin, evt, delay)` se déclenche donc `delay` seconde après l'occurrence de `begin`, mais seulement si `evt` n'a pas été détecté entre temps.

### 7.2.4 Capture des occurrences

Nous l'avons déjà dit en introduction l'objectif des descripteurs d'événements n'est pas seulement d'indiquer si oui on non un événement donné s'est produit, mais aussi de passer suffisamment d'information au reste de la règle réactive pour prendre la bonne décision concernant l'adaptation de l'application.

Pour cela, nous introduisons un mécanisme qui permet de *capturer* les occurrences d'événements primitifs détectées et de les rendre utilisables dans les autres parties de la règle réactive (condition et action). Ce mécanisme est similaire à l'utilisation des variables spéciales `$1`, `$2`, etc. dans les systèmes de détection d'expressions régulières, mais utilise des noms symboliques spécifiés explicitement plutôt que des chiffres associés automatiquement aux groupes de parenthèses.

Pour capturer une occurrence d'événement primitif, il suffit de préfixer son descripteur par un symbole suivi de deux points « `:` » : `symbole:type_événement(args | pred)`. Par exemple :

```
meth:message-received($c/interface::*)
device:appears(sys://devices/usb/*)
```

Lorsqu'un événement – éventuellement composite – a été détecté, tous les événements primitifs qui ont été marquées de cette façon et qui ont contribué à la détection de l'événement global sont rendus visibles au reste de la règle réactive sous la forme de variables `$symbole.champ` :

```
when e:subcomponent-added($target)
if (count($target/child::*[started(.)]) > 10)
do { stop($e.sub-component); }

when e:subcomponent-removed($target)
if (count($target/child::*[started(.)]) < 10)
do { start($target/child::*); }
```

Les deux règles ci-dessus s'assurent que le composite `$target` ne contient pas plus de 10 composants actifs à la fois. `$target` est une variable spéciale qui désigne le composant cible sur lequel une politique d'adaptation est attachée, et donc le composant SAFRAN que les règles doivent adapter. Le corps de l'action de la première règle référence la variable `$e.sub-component`, qui correspond au champ `sub-component` de l'occurrence de l'événement `e`, capturée par `e:subcomponent-added($target)`, et donc au nouveau sous-composant effectivement ajouté à `$target`.

```
when never(noise:changed(phys://environment/sound#level),
           changed(phys://environment/sound#level | abs($noise.new-value - $new-value) > 10,
           5))
do {
  volume := $noise.new-value * pref://multimedia/audio#volume_factor;
  set-value($target/@volume, $volume);
}
```

Celle-ci, destinée à un composant « lecteur multimédia » ajuste le volume sonore du lecteur en fonction du bruit ambiant et des préférences de l'utilisateur. Cet ajustement ne se fait que si le bruit ambiant change de plus de 10% pendant au moins 5 secondes.

## 7.3 Structure des politiques d'adaptation

### 7.3.1 Introduction

Une politique d'adaptation est structurée sous la forme d'une séquence de règles réactives de type ECA. Ces règles permettent de désigner les circonstances qui doivent déclencher une adaptation (quand adapter ?), et les modifications à appliquer pour rendre l'application mieux adaptée à ces nouvelles circonstances (comment adapter ?). Une politique est en réalité un simple conteneur de règles réactives

destiné à regrouper un ensemble cohérent de telles règles, qui peuvent ensuite être manipulées comme un tout. Chaque politique d'adaptation individuelle implémente un scénario d'adaptation plus ou moins complexe. Par exemple, l'adaptation des paramètres d'un flux multimédia aux performances du réseau et de l'hôte, que l'on peut considérer comme un scénario unique, nécessite plusieurs règles d'adaptation qui coopèrent pour appliquer les bonnes modifications aux moments appropriés.

Les politiques d'adaptation sont décrites dans de simples fichiers texte portant l'extension `.policy`, qui peuvent contenir trois types de définitions différentes :

1. définitions de *fonctions* et d'*actions* FScript[4] ;
2. définitions de *règles* réactives ;
3. définitions de *politiques* d'adaptation.

Chacun de ces types de définitions dispose de son propre espace de nommage. Il est possible d'avoir par exemple une fonction et une règle portant le même nom sans risque de conflits.

### 7.3.2 Définition de règles réactives

La définition d'une règle d'adaptation permet de donner un nom symbolique à un triplet ECA, qui pourra ensuite être utilisé dans la définition d'une ou plusieurs politiques. Il s'agit purement d'un mécanisme de ré-utilisation de code, et il n'est pas nécessaire de définir (et donc de nommer) une règle pour pouvoir l'utiliser : la syntaxe des politiques d'adaptation supporte aussi l'utilisation de règles anonymes insérées directement dans la politique. Nommer une règle, même si elle n'est utilisée qu'une seule fois peut cependant aider à la compréhension d'un politique en rendant plus explicite l'intention de son programmeur.

La syntaxe d'une définition de règle est :

```
rule <name> = {
  when <event_desc>
  if <guard>
  do <action>
}
```

comme par exemple

```
rule enforce_minimum = {
  when e:subcomponent-removed($target)
  if (count($target/child::*) < 10)
  do {
    start($target/child::*);
  }
}
```

- `name` est un identifiant servant à nommer la règle. Dans l'exemple, le nom de la règle est `enforce_minimum`.
- `event_desc` est un descripteur d'événement (partie $E$ de ECA), primitif ou composite, conforme au langage décrit dans la section 7.2. Dans l'exemple, il s'agit de `e:subcomponent-removed($target)`, qui détecte le retrait d'un sous-composant du composite `$target` et capture cet événement dans la variable `$e`.
- `guard` est la condition ($C$), ou garde. Il doit s'agir d'une expression booléenne FPath (donc sans effet de bord). Cette partie est optionnelle, et est considérée toujours *vrai* si elle est omise. La condition de la règle ci-dessus est `(count($target/child::*) < 10)` qui teste si `$target` a moins de 10 sous-composants.

[4] Les syntaxes de FPath et FScript sont légèrement étendues dans le cadre des politiques d'adaptation pour permettre de référencer directement le contexte d'exécution WildCAT. Il est ainsi possible d'écrire directement `sys ://memory#free` au lieu de passer par une fonction `context("sys ://memory#free")`. Il s'agit purement de « sucre syntaxique » destiné à alléger l'écriture et la lecture des politiques d'adaptation.

– Enfin, `action` décrit sous la forme d'un programme FScript la reconfiguration à appliquer pour adapter l'application. Dans le cas de l'exemple, cette action consiste à démarrer tous les sous-composants de `$target`.

Une règle devant former un tout, et étant destinée *in fine* à être attachée à un composant Fractal, ses différentes parties ne sont pas complètement indépendantes les unes des autres. En particulier, il existe des contraintes concernant les variables libres utilisables dans les différents parties d'une règle :

– N'importe quelle partie d'une règle peut référencer la variable spéciale `$target`, mais aucune ne peut la redéfinir. Cette variable sert de paramètre implicite à une règle et représente le composant cible à adapter. Sa valeur est définie automatiquement par le système lorsqu'une politique est attachée à un composant, et est donc la même pour toutes les règles de toutes les politiques attachées à ce composant.
– Le descripteur d'événement (partie $E$) ne peut référencer que la variable `$target` et les variables qu'il définit lui-même par capture.
– La guarde (partie $C$) peut référencer `$target` et toutes les variables capturées par le descripteur d'événements. Cette partie de la règle étant une expression booléenne simple, elle ne peut pas définir ou redéfinir de variables.
– Enfin, la partie action ($A$) peut référencer `$target` ainsi que les variables définies par capture et toutes les variables locales définies dans le corps de l'action elle-même (après qu'elles aient été définies bien entendu).

### 7.3.3 Définition de politiques d'adaptation

Une politique d'adaptation est simplement un ensemble ordonné de règles d'adaptation auquel on donne un nom :

```
policy <name> = { <rules> }
```

Il existe trois façons différentes d'indiquer les règles qui font partie d'une politique :

**Description en ligne.** La plus simple consiste simplement à décrire directement la règle (*inline*) :

```
rule {
  when <event_desc>
  if <guard>
  do <action>
}
```

**Référence.** La deuxième consiste à référencer par son nom une règle définie ailleurs grâce à la syntaxe décrite précédemment : `rule <rule_name>;`.

**Inclusion.** Enfin, la dernière permet d'inclure simplement l'ensemble des règles d'une autre politique d'adaptation : `include <policy_name>;`. Cette construction permet de réutiliser un ensemble de règles dans plusieurs politiques et de l'étendre en ajoutant ensuite d'autres règles.

Voici un exemple complet d'une politique d'adaptation :

```
policy disconnected-mode = {
  rule {
    when disappears(sys://network/interfaces/eth0)
    do {
      mc := new("deferred-messages");
      set-meta($target, $mc);
      start($mc);
    }
  }
  rule {
```

```
    when appears(sys://network/interfaces/eth0)
    do {
      if ($target/metalink::*) then {
         stop($target/metalink::*);
         unset-meta($target);
      }
    }
  }
}
```

## 7.4 Modèle d'exécution des politiques d'adaptation

### 7.4.1 Introduction

Maintenant que nous avons vu la structure (statique) des politiques d'adaptation, nous décrivons le détail de leur modèle d'exécution (dynamique) qui leur permet d'adapter effectivement une application Fractal. Comme cela était déjà le cas pour la partie structurelle, la sémantique opérationnelle des différentes parties des règles d'adaptation ($E$, $C$ et $A$) a déjà été présentée. Nous nous concentrerons donc sur l'intégration de ces éléments et leurs interactions sans plus entrer dans le détail de leur dynamique individuelle.

Pour cela, nous commencerons tout d'abord par présenter la vision globale de l'intégration des politiques d'adaptation dans les applications Fractal (Section 7.4.2) et les différentes étapes de leur cycle de vie. Nous décrirons ensuite la dynamique des politiques en suivant une approche « *bottom-up* » : dans un premier temps la dynamique d'une politique d'adaptation prise individuellement (Section 7.4.3), puis les interactions entre politiques d'un même composant (Section 7.4.4) pour finir par les interactions entre composants adaptatifs eux-même au sein d'une application (Section 7.4.5).

### 7.4.2 Cycle de vie des politiques

Notre objectif final étant de faciliter le développement d'applications adaptatives, SAFRAN doit s'intégrer le plus simplement possible dans le cycle de développement standard.

Les applications à base de composants sont habituellement conçues en suivant un certain nombre d'étapes (cf. Section 4.7) :

1. *spécification* de l'architecture de l'application ;
2. *implémentation* de chacun des composants identifiés, soit en réutilisant des composants existants soit en en créant de nouveaux ;
3. *déploiement* de l'application résultante, ce qui inclut son installation, sa configuration, et enfin son exécution.

Ce schéma fonctionne dans le cas où les seuls artefacts à prendre en compte sont les composants eux-mêmes, mais SAFRAN introduit justement un nouveau type d'artefacts qui participent à l'application, sous la forme des politiques d'adaptation. Celles-ci ont leur propre cycle de développement, qui doit s'intégrer le plus naturellement possible dans le cycle existant. Notons que du point de vue de l'adaptation, ce qui différencie les différentes phases ci-dessus est avant tout une connaissance de plus en plus précise des conditions d'exécutions exactes dans lesquelles l'application fonctionnera.

Le cycle de développement des politiques elles-mêmes est caractérisé par les étapes suivantes :

**Définition de la politique.** Cette étape correspond à la programmation de la politique grâce au langage dédié de SAFRAN. Les politiques sont stockées dans de simples fichiers textes qui seront chargés lors des étapes suivantes. C'est le découplage fort – spatial *et* temporel – introduit par SAFRAN entre le code métier des composants et les politiques qui permet d'effectuer cette première étape à n'importe quel moment du cycle de développement de l'application elle-même, selon la nature de

l'adaptation à réaliser. Pendant la spécification de l'architecture de l'application, on peut définir des politiques d'adaptation globales, comme par exemple pour la répartition de charge ; lors de l'implémentation des composants, certains peuvent nécessiter des politiques spécifiques, comme par exemple l'ajout d'un cache transparent ; une fois l'application installée et configurée pour un hôte particulier, de nouvelles politiques peuvent être définies pour tirer partie des capacités spécifiques de ce dernier et de l'utilisation envisagée ; enfin, pendant l'exécution, les besoins peuvent évoluer et demander la création de nouvelles politiques, par exemple pour mieux gérer une montée en charge imprévue.

**Attachement.** Une fois la politique définie, elle doit ensuite être attachée à un ou plusieurs composants de l'application. Puisque les composants Fractal n'existent réellement qu'à l'exécution, cet attachement se fait forcément dynamiquement. L'attachement peut être déclenché soit par un programme (par exemple une console d'administration), en utilisant l'interface `AdaptationController`, soit par une autre politique, qui peut utiliser l'action FScript `attach()` (cf. Section 6.4.4) dans le cadre d'une action d'adaptation. Dans tous les cas, une fois que la politique est attachée à un composant, elle prend en charge l'adaptation de celui-ci.

**Exécution.** C'est la phase la plus importante du cycle de vie des politiques, pendant laquelle elles adaptent les composants auxquels elles sont attachées en réagissant aux modifications du contexte d'exécution. La sémantique précise de l'exécution des politiques et de leurs interactions est l'objet des sections suivantes.

**Détachement.** Enfin, l'association entre une politique et un composant étant dynamique, une politique peut être détachée d'un composant à tout moment. De la même manière que pour l'attachement, cette étape peut être déclenchée soit directement en utilisant l'interface `AdaptationController` du composant concerné, soit par l'exécution d'une autre politique d'adaptation en utilisant l'action FScript `detach()`.

Bien entendu, les phases d'attachement, exécution et détachement peuvent être répétées et une même définition de politique peut être « instanciée » un nombre quelconque de fois pour être utilisée avec plusieurs composants. Même si elles sont issues de la même définition, deux politiques associées à deux composants différents sont considérées comme différentes et ne partage aucun état (le seul « état » d'une politique étant représenté par la variable `$target` qui identifie le composant cible).

### 7.4.3 Comportement individuel des politiques

Nous décrivons maintenant la dynamique d'une politique d'adaptation attachée à un composant Fractal. Nous supposerons dans un premier temps qu'une seule politique $P$ est attachée au composant cible $C$ ; le cas plus complexe des politiques multiples interagissant sera traité dans la section suivante.

La sémantique d'une politique d'adaptation résulte de la combinaison de ses règles ECA. Si le paradigme ECA peut sembler simple au premier abord (*lorsque* événement, *si* condition, *exécuter* action), il regroupe en réalité de très nombreuses variantes, en fonction de la sémantique individuelle des différentes parties des règles, des modes de couplage utilisés (en particulier temporel), etc. La sémantique que nous avons choisie pour nos règles d'adaptation dans le cadre de SAFRAN doit donc être considérée comme un point particulier dans l'espace des sémantiques possibles pour ECA, qui nous semble le plus approprié pour notre problème particulier.

Bien que le comportement d'une politique d'adaptation ne concerne à strictement parler que l'étape d'*exécution* décrite ci-dessus, il est important de comprendre les pré- et post-traitements associées aux étapes d'attachement et de détachement.

**Algorithme.** L'algorithme complet décrivant le comportement des politiques peut être résumé ainsi :

**A1.** [Initialisation.] Lors de l'attachement de la politique $P$ au composant $C$, le contrôleur d'adaptation de $C$ extrait de $P$ l'ensemble des événements primitifs mentionnés dans les règles de $P$. Chaque descripteur est traité selon son type :

**A1.1.** [Configuration de WildCAT.] S'il s'agit d'un événement *exogène*, le contrôleur s'abonne auprès de WildCAT afin d'être notifié à chaque occurrence de cet événement.

**A1.2.** [Instrumentation.] S'il s'agit d'un événement *endogène*, le système se charge d'instrumenter le programme de façon appropriée pour que $C$ soit notifié à chaque occurrence de cet événement.

**A2.** [Attente.] La politique d'adaptation reste passive dans l'attente de la réception d'un événement.

**A3.** [Sélection.] Lorsqu'une occurrence d'événement est reçue, le contrôleur d'adaptation de $C$ commence par déterminer l'ensemble des règles qui doivent en être notifiées. L'occurrence est envoyée tour à tour à chacune de ces règles candidates, qui se contentent de déterminer si l'événement les active ou pas.

**A4.** [Reconfiguration.] Les actions conditionnelles de chacune des règles encore actives à cette étape sont combinées et exécutées en une seule « transaction » de reconfiguration. La combinaison est une simple concaténation des couples *Condition* / *Action* de ces règles, dans l'ordre d'apparition (textuel) à l'intérieur de la définition de la politique. Une fois l'opération d'adaptation combinée appliquée (quel qu'en soit le résultat), on retourne ensuite à l'étape A2 en attente d'un nouvel événement.

**A5.** [Détachement.] Avant de détacher une politique d'adaptation, le contrôleur d'adaptation se désabonne de tous les événements correspondants (exogènes et endogène), et termine les éventuelles reconfigurations déjà en cours.

Quelques points de cet algorithme méritent d'être discutés plus en détail.

**Initialisation.** L'attachement d'une politique $P$ à un composant cible $C$ se fait toujours, directement ou non, par l'intermédiaire du contrôleur d'adaptation de $C$, i.e. son interface `adaptation-controller`. Lorsque $C$ est attachée à $P$, ce contrôleur commence par s'assurer que tous les événements nécessaires à l'exécution de $P$ lui seront bien envoyés. Pour cela, il interroge les différentes règles de $P$ et en extrait tous les descripteurs d'événements primitifs. S'il s'agit d'un événement *exogène* – lié au contexte d'exécution de l'application – le contrôleur contacte WildCAT et demande à être notifié de l'événement correspondant. Par exemple, si l'une des règles de $P$ mentionne l'événement `changed(sys://storage/memory#free)` le contrôleur demande à WildCAT de le notifier à chaque modification de l'attribut en question. Le traitement des événements endogènes est similaire, le contrôleur s'abonnant auprès des composants appropriés pour recevoir ces événements[5]. Pour chaque descripteur d'événement primitif, le contrôleur d'adaptation garde en mémoire la ou les règles qui le mentionne dans une « table de routage ». Ainsi, lorsqu'une occurrence d'événement lui arrive pendant la phase d'exécution proprement dite, il sait quelles règles en sont à l'origine.

**Exécution.** Tout comme le processus d'adaptation lui-même, la phase d'exécution des politiques est purement réactive, c'est-à-dire qu'une fois l'initialisation terminée, le contrôleur d'adaptation se contente d'attendre de recevoir des notifications d'événements.

- Lorsqu'une telle occurrence arrive, il utilise la table de routage mentionnée ci-dessus pour déterminer un ensemble initiale de règles candidates. Chacune de ces règles détermine alors si cette occurrence déclenche l'événement – éventuellement composite – qui lui correspond. Celles dont l'événement ne correspond pas sont supprimées de l'ensemble des candidates.
- La seconde phase correspond à l'exécution conditionnelle des *Actions*, qui vont effectivement adapter l'application. Pour cela, les couples condition / action des règles candidates sont combinées pour ne former qu'une seule action composite. En effet, une politique est censée implémenter un scénario d'adaptation cohérent. Si chaque action était traitée séparément, alors en cas de problème lors de l'exécution de certaines le système pourrait se retrouver « à moitié adapté », ce

[5] Dans l'implémentation actuelle, l'instrumentation des composants, et donc la génération des événements endogènes, est globale à un composant et définie au moment de son instanciation. Cette approche « tout ou rien » simple mais assez coûteuse en performances pourrait être améliorée dans le futur par l'utilisation des techniques de réflexion partielle introduites par Reflex [Tanter et al., 2003].

qui peut être pire que de rien faire. En regroupant toutes les actions à appliquer en une seule, nous utilisons les propriétés de FScript (atomicité, consistance...) pour nous assurer que l'adaptation voulue par le programmeur de la politique est bien appliquée de façon cohérente. Concrètement, la composition des différentes réactions se fait tout simplement par concaténation : l'action composite est la séquence des actions conditionnelles de toutes les règles à appliquer. L'ordre utilisé pour la concaténation est simplement l'ordre (textuel) dans lequel les règles ont été définies dans la politique, ce qui évite d'avoir à introduire de nouveaux mécanismes dans le langage. Conceptuellement, cela revient à créer une nouvelle action FScript dont le corps serait de la forme `{ if (condition1) then { action1(); } if (condition2) then { action2(); } ... }`. Cette action globale est donc exécutée de façon atomique par FScript afin d'adapter le composant cible. Si une erreur quelconque survient pendant l'exécution de l'action, ou que l'action résulte en une nouvelle configuration invalide, FScript annule la reconfiguration et s'assure que les composants affectés retournent dans leur état initial. Dans tous les cas, le contrôleur d'adaptation retourne alors dans son état passif en attendant un nouvel événement.

**Récursion.** Afin d'éviter les problèmes de récursion infinie (l'exécution d'une action génère un événement qui déclenche une action, qui génère un événement...), les événements endogènes générés pendant l'exécution d'une action sont ignorés, et ne peuvent donc pas déclencher de réactions.

**Parallélisme et synchronisation.** La partie détection des événements est toujours active, même si une reconfiguration est en cours. Par contre, le test des conditions ne peut se faire que lorsque le système est dans un état stable (critère d'isolation), et une seule reconfiguration à la fois peut être active sur un composant donné. Si une occurrence d'événement est reçue pendant qu'une reconfiguration est en cours et qu'elle déclenche l'activation d'une ou plusieurs règles, le traitement de ces dernières est mis en attente jusqu'à la fin de la reconfiguration en cours. Si plusieurs activations sont mises en attente, elles sont ensuite traitées dans l'ordre chronologique de leur arrivée (FIFO).

**(A)synchronicité.** Selon la nature (endogène ou exogène) de l'occurrence d'événement qui déclenche effectivement l'adaptation, l'exécution de la réaction peut bloquer ou non l'exécution de l'application elle-même. Si l'événement déclencheur est exogène, il désigne une évolution du contexte d'exécution, qui est par nature asynchrone par rapport à l'exécution de l'application. Dans ce cas, la reconfiguration s'effectue en parallèle du flot d'exécution normal de l'application adaptée. En revanche, si l'événement déclencheur est endogène, sa détection bloque le flot d'exécution de l'application pendant l'exécution de la réaction.

**Désactivation.** La dernière phase importante dans le cycle de vie d'une politique est sa désactivation, préliminaire à son détachement du composant cible. Lorsque le détachement d'une politique est demandé, le contrôleur d'adaptation commence par ignorer tous les nouveaux événements qui lui arrivent, afin de ne pas déclencher de nouvelles reconfigurations. Il se désabonne ensuite de tous les événements auxquels il était abonné. Dans le cas des événements exogènes, il lui suffit de prévenir WildCAT ; pour les événements endogènes, il indique simplement aux composants concernés qu'il n'est plus intéressé par certains événements. Enfin, le contrôleur d'adaptation attend que toutes les reconfigurations en cours se terminent avant de retirer effectivement la politique du composant cible.

### 7.4.4 Interactions entre politiques d'un même composant

Puisque la section précédente ne traite que du cas particulier où une seule politique est présente à la fois, nous devons maintenant décrire le comportement global d'un composant adaptatif lorsque *plusieurs* politiques d'adaptation s'y trouvent attachées en même temps. En effet, l'interface `AdaptationController` permet d'attacher et de détacher un nombre quelconque de politiques à n'importe quel moment.

On peut identifier deux alternatives fondamentales :

– La première consiste à *sélectionner* – selon un critère à définir – l'une des politiques associées au composant, et à ne considérer que celle-ci. Ce choix nous ramène alors au cas précédent d'une seule politique active à la fois, même lorsque plusieurs sont attachées.
– La seconde consiste à l'inverse à composer le comportement de toutes les politiques attachées. Cette composition peut se faire de différentes manières :
  – en *fusionnant* toutes les règles des différentes politiques pour n'en former qu'une seule (ce qui nous ramène encore une fois au cas précédent) ;
  – ou en traitant les politiques *en séquence*, selon un ordre à déterminer.

  Étant donné qu'une seule transaction de reconfiguration peut être active à la fois, il n'est pas possible de traiter les politiques en parallèle, du moins pas en ce qui concerne les parties *Action* de leurs règles.

La première stratégie, qui consiste à ne retenir qu'une seule politique au détriment de toutes les autres, nous semble inappropriée. En effet, un tel choix empêcherait de traiter pour un même composant plusieurs scénarios d'adaptation orthogonaux (une politique pour la sécurité, une pour les performances, etc.), à moins de créer manuellement des politiques combinant les différents scénarios, ce qui va à l'encontre de nos critères de modularité et de dynamicité de l'aspect d'adaptation.

La fusion des règles formant les différentes politiques ne fonctionne pas non plus. Si les politiques ont été définies séparément, c'est qu'elles traitent de scénarios d'adaptation différents, et si on fusionne leurs règles, deux règles issues à l'origine de deux politiques différentes mais activées par un même événement verront leurs *Actions* traitées comme une seule transaction. Dans ce cas, l'échec d'une seule empêchera l'exécution de l'autre. Une politique mal écrite (avec un *Evénement* trop général et une *Action* qui échoue toujours par exemple) pourrait alors interférer avec l'exécution des autres politiques d'un composant cible donné. Encore une fois, ce choix va à l'encontre de notre critère de modularité de l'aspect d'adaptation.

La solution que nous avons retenue est donc basée sur le traitement en séquence des politiques attachées à un composant cible. Plus précisément, la première étape de détection des événements et d'activation des règles se fait en considérant toutes les règles de toutes les politiques, dans un ordre quelconque. Une fois qu'un ensemble de règles a été activé, la suite de leur exécution (test de la condition et exécution de l'action) se fait politique par politique. Si deux règles issues des politiques $P_1$ et $P_2$ sont activées, la condition $C_{P_2}$ doit être testée *après* que l'action $A_{P_1}$ ait été exécutée : sinon, on risque d'exécuter une reconfiguration à partir d'un test éventuellement rendu obsolète par $A_{P_1}$. L'ordre d'exécution est déterminé par l'ordre d'attachement des politiques (FIFO), c'est-à-dire que la politique la plus « ancienne » (attachée au composant depuis le plus de temps) est exécutée la première. En effet, lorsqu'on attache une politique d'adaptation $P_1$ à $C$, le composant résultant doit être considéré comme un tout, une « boîte noire » qui encapsule $C$ *et* $P_1$. Lorsque $P_2$ est attachée ensuite, on doit considérer qu'elle s'applique à $C + P_1$, et pas seulement à $C$ ; puisque $P_2$ adapte $C + P_1$, elle doit donc être exécutée après $P_1$. Si l'exécution des règles issues d'une des politiques génère une erreur (et est donc annulée), le traitement des politiques suivantes continue tout de même (pour la même raison qui nous a fait rejeter l'approche par fusion). Pour résumer, l'algorithme est donc le suivant :

**B1.** [Activation des règles.] Lorsqu'un événement $E$ est reçu par le contrôleur d'adaptation du composant cible $C$, ce dernier l'envoie à toutes les règles de toutes les politiques attachées à $C$. Il en résulte un ensemble global de règles activées.

**B2.** [Exécution sérialisée.] Pour chaque politique $P_i$ attachée à $C$, triées dans leur ordre d'attachement (FIFO) :

**B2.1.** [Construction de la réaction.] Le contrôleur détermine tout d'abord la réaction de cette politique en concaténant les conditions et actions de toutes ses règles activées, dans l'ordre de leur définition.

**B2.2.** [Exécution de la réaction.] L'action composite FScript résultante est exécutée dans une reconfiguration transactionnelle.

Si une erreur se produit pendant l'exécution des reconfigurations demandées par $P_i$, le contrôleur continue le traitement et passe à $P_{i+1}$.

### 7.4.5 Interactions entre composants adaptatifs

Le dernier point qui nous reste à voir concerne les interactions *entre composants* adaptatifs au sein d'une même application. Lorsqu'un événement – exogène ou endogène – se produit au sein d'une application, le système SAFRAN doit en notifier tous les composants dont au moins un politique d'adaptation est concernée par cet événement. L'ordre dans lequel les différents composants sont notifiés détermine l'ordre d'exécution de leurs réactions.

L'un des concepts fondamentaux de la programmation par composants est l'encapsulation forte, sous la forme de composants « boîte noire » : lorsqu'un composite $C$ utilise un sous-composant $C'$, il n'a pas à connaître les détails d'implémentation de ce dernier, seulement son interface fonctionelle. Considérant que l'*aspect d'adaptation*, représenté dans SAFRAN par les politiques réactives, fait partie intégrante de l'implémentation d'un composant adaptatif, nous choisissons de donner la priorité dans l'ordre d'exécution aux composants les plus imbriqués, l'ordre de traitement de composants au même niveau d'imbrication étant indéterminé. Ainsi, chaque composant adaptatif a la possibilité de s'adapter lui-même (boîte noire) avant que les composites qui l'utilisent puissent interférer avec son fonctionnement (dont l'adaptation fait partie intégrante).

Le critère d'*isolation* des reconfigurations (lié à la modularité des politiques d'adaptation) nous impose que deux reconfigurations qui se chevauchent ne peuvent pas s'exécuter en même temps, et doivent donc être sérialisées. Deux reconfigurations $R_1$ et $R_2$ se chevauchent si : elles modifient certains éléments (composants, connexions ou attributs) en commun, *ou* $R_1$ modifie des éléments inspectés par $R_2$, *ou* $R_2$ modifie des éléments inspectés par $R_1$. Dans tous ces cas, l'exécution en parallèle des deux reconfigurations risque de rendre visibles à $R_1$ les états intermédiaires potentiellement inconsistants résultant de l'exécution partielle de $R_2$ (et inversement).

Nous devons donc détecter *avant* l'exécution d'une reconfiguration $R_2$ si son exécution risque de chevaucher une reconfiguration $R_1$ déjà en cours d'exécution. Si c'est le cas, $R_2$ doit être mise en attente et exécutée uniquement une fois $R_1$ terminée. Malheureusement, il n'est pas possible de déterminer précisément l'ensemble des éléments affectés par une reconfiguration sans l'exécuter (ou au moins la simuler). Nous pouvons cependant utiliser un critère d'isolation plus fort mais plus facile à garantir. En particulier le cas extrême (qui correspond à l'utilisation d'un verrou d'exclusion global) est de n'autoriser qu'une seule reconfiguration à la fois pour toute l'application. C'est le choix effectué par l'implémentation actuelle, pour des raisons de simplicité. D'autres pistes correspondant à l'utilisation de verrous plus fins sont à l'étude pour les versions futures de l'implémentation.

## 7.5 Conclusion

Dans ce chapitre, nous avons décrit le langage dédié SAFRAN permettant de programmer des politiques d'adaptation réactives destinées à être associées (tissées) à des composants Fractal. Pour cela, nous avons utilisé les différentes contributions décrites dans les chapitres précédents, ainsi qu'un langage pour la spécification et la détection d'événements.

Nous avons décrit la dynamique des politiques d'exécution en partant du cas simple d'une seule politique attachée à un seul composant. Nous avons ensuite montré comment ce cas particulier pouvait être étendu à un ensemble dynamique de politiques sur un unique composant puis à une application complète, en accord avec les principes de conception des applications adaptatives en général et de SAFRAN en particulier. Pour résumer, lorsqu'un événement « intéressant » (i.e. succeptible de déclencher une adaptation) se produit, SAFRAN le diffuse à chacun des composants adaptatifs concernés, en commençant par les plus imbriqués. Pour chacun de ces composants, les règles de leurs politiques d'adaptation que l'événement active sont alors exécutées, en isolant dans des transactions de reconfiguration les actions issues de différentes politiques, et en traitant les politiques dans leur ordre d'attachement.

Il en résulte un système complet et cohérent permettant de développer des politiques d'adaptation *séparément* des composants métier, aussi bien sur le plan spatial que temporel. Associées (tissées) dynamiquement aux composants Fractal d'une application, les politiques d'adaptation SAFRAN pemettent de rendre ces composants *adaptatifs*.

# Chapitre 8

# Développement d'applications adaptatives avec SAFRAN

## Sommaire



## 8.1 Introduction

Ce chapitre présente l'utilisation concrète de SAFRAN pour le développement d'applications adaptatives, et l'illustre à travers quelques exemples qui nous servent ensuite à évaluer notre système.

Nous commençons par décrire le modèle de développement d'applications adaptatives et les outils fournis aux programmeurs par SAFRAN pour supporter ce développement. Nous présentons ensuite rapidement notre méthodologie et nos critères d'évaluation, avant de montrer quelques exemples concrets d'applications adaptatives développées avec SAFRAN. Enfin, nous concluons en évaluant l'apport de SAFRAN sur ces applications, validant ainsi nos travaux.

## 8.2 Modèle et outils de développement

Nous donnons maintenant quelques détails pratiques sur l'utilisation de SAFRAN du point de vue du programmeur d'application adaptatives.

La figure 8.1 résume le processus de développement d'une application adaptative avec SAFRAN :

- L'application *adaptable* est distribuée sous la forme d'un paquetage (fichier `.jar`) contenant à la fois l'implémentation des composants (fichiers `.class`) et les fichiers ADL décrivant son architecture (fichiers `.fractal`). Ces composants peuvent être développés spécifiquement pour l'application ou être des composants sur l'étagère (COTS) réutilisés tels quels.
- Le lancement de l'application se fait presque comme pour une application Fractal normale, la seule différence étant que les sous-systèmes de SAFRAN (WildCAT en particulier) doivent être initialisés auparavant.
- Pendant l'exécution, il est alors possible d'attacher et de détacher des politiques d'adaptation SAFRAN (fichiers `.policy`) aux composants de l'application. Ces politiques peuvent avoir été développées à n'importe quel moment pendant le développement et l'exécution de l'application.

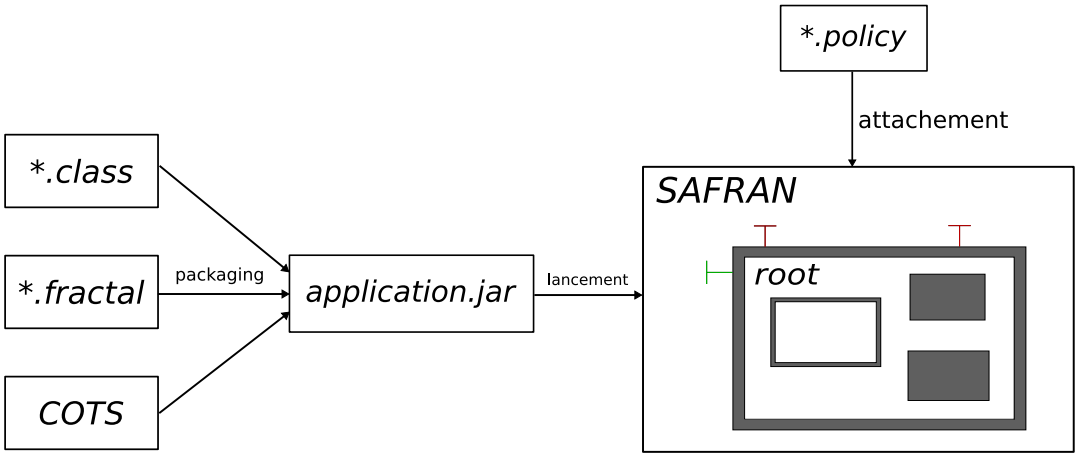


FIG. 8.1 – Processus de développement d'applications adaptatives avec SAFRAN.

L'interface de programmation présentée par SAFRAN aux programmeurs peut être considérée selon trois points de vue, qui correspondent aux différentes phase du cycle de développement de l'application : *(i)* pendant la phase d'implémentation, comment développer un composant adaptatif SAFRAN ; *(ii)* au démarrage de l'application adaptative, comment configurer et démarrer le support d'exécution (*runtime*) de SAFRAN et de ses sous-systèmes ; et enfin *(iii)* pendant l'exécution de l'application, comment manipuler SAFRAN, et en particulier l'attachement et le détachement de politiques d'adaptation.

### 8.2.1 Programmation de composants adaptatifs

Puisque l'objectif de SAFRAN est de séparer le plus possible le code spécifique à l'aspect d'adaptation du code métier, l'implémentation des composants eux-mêmes est quasi-identique à celle de composants Fractal standards. Les classes Java correspondant aux composants primitifs d'une application n'ont pas besoin d'implémenter une quelconque interface, ou de suivre des conventions de codage pour pouvoir être rendues adaptatives. N'importe quel composant Fractal, même existant, peut techniquement être utilisé par SAFRAN et rendu adaptatif par l'attachement de politiques appropriées. Bien entendu, ces politiques ne peuvent adapter un composant que dans la mesure où il propose des possibilités de reconfiguration, par exemple en exposant des paramètres (`attribute-controller`).

Concrètement, la seule chose qui distingue un composants adaptatif SAFRAN d'un composant Fractal « standard » est l'utilisation de contrôleurs spécifiques, spécifiés lors de la construction du composant.

Julia, l'implémentation de référence de Fractal sur laquelle SAFRAN est basée, utilise la notion d'*alias* pour désigner une configuration de contrôleur prédéfinie. Par défaut, les alias `primitive` et `composite` correspondent à des composants Fractal standard. SAFRAN définit deux nouveaux alias, `adaptivePrimitive` et `adaptiveComposite` qui correspondent à des configurations de contrôleurs intégrant toutes les extensions de SAFRAN : le MOP, les contraintes architecturales, et bien sûr le contrôleur d'adaptation. Ainsi, pour rendre adaptatif un composant Fractal définit avec l'ADL, il suffit de modifier sa définition de la façon suivante :

```
<!-- Définition standard du composant -->
<definition name="examples.Printer">
  <interface name="printer" role="server" signature="examples.Printer"/>
  <content class="examples.PrinterImpl"/>
  <!-- Implicitement: <controller desc="primitive"/> -->
</definition>

<!-- Définition d'un composant adaptatif -->
<definition name="examples.Printer">
  <interface name="printer" role="server" signature="examples.Printer"/>
  <content class="examples.PrinterImpl"/>
  <controller desc="adaptivePrimitive"/>
</definition>
```

Bien entendu, il est aussi possible d'instancier explicitement un composant adaptatif à partir de Java grâce à l'interface `GenericFactory`, en utilisant l'alias approprié.

De cette façon, il est possible de choisir pour chaque composant d'une application s'ils doivent être adaptatifs ou non. De même, si par défaut les alias `adaptive*` incluent les trois contrôleurs `metalinkcontroller`, `constraints-controller` et `adaptationcontroller`, la conception modulaire de SAFRAN rend très simple de définir de nouveaux alias incluant uniquement les extensions nécessaires à un composant, voire d'autres extensions qui pourront être définies ultérieurement. De même, il est possible de redéfinir les alias standards `primitive` et `composite` pour que tous les composants d'une application soient par défaut adaptatifs.

### 8.2.2 Configuration et initialisation du système

Tous les composants de l'application adaptative ayant été développés et « marqués » adaptatifs ou non, l'application est prête à être démarrée. Le déploiement et le démarrage d'une application SAFRAN est légèrement plus complexe que pour une application Fractal normale, car il faut indiquer au système où trouver les différents fichiers nécessaires à son exécution :

- les fichiers contenant les définitions des politiques d'adaptation ;
- les fichiers contenant les définitions de nouvelles procédures FScript (fonctions et actions) ;
- et enfin les fichiers de configuration nécessaires au système WildCAT.

Pour cela, nous utilisons la notion de *propriété système* définie par Java, qui permet de spécifier ce genre de paramètres de configuration d'une application. Spécifiquement, SAFRAN utilise les propriétés suivantes pour localiser les fichiers dont il a besoin :

- `org.obasco.safran.policies.path` doit être une liste de répertoires (séparés par deux points « : ») où se trouvent les fichiers de définition des politiques d'adaptation. Ces fichiers doivent porter l'extension `.policy`, comme par exemple `memory-usage.policy`.
- `org.obasco.safran.fscript.path` indique de la même manière où trouver les fichiers `.fscript` contenant les définitions de procédures FScript.
- `org.obasco.safran.wildcat.path` indique où WildCAT peut trouver les fichiers XML de définition des domaines contextuels.
- Enfin, les propriétés `org.obasco.safran.wildcat.domain.`*`name`* associent à un nom de domaine contextuel (*`name`*) le fichier contenant sa définition. Par exemple, la propriété `org.obasco.safran.`

`wildcat.domain.sys` pourrait avoir pour valeur `linux.xml`, indiquant que la définition du domaine `sys ://` se trouve dans le fichier `linux.xml` (relativement au chemin de recherche).

Une fois ces propriétés fixées, le système SAFRAN doit être initialisé avant de charger et lancer l'application elle-même. Pour cela, il suffit d'invoquer la méthode statique

```
org.obasco.safran.Safran.initialize();
```

puis de démarrer l'application normalement, en instanciant son composant racine et en invoquant son point d'entrée principal.

### 8.2.3 Contrôle à l'exécution

Une fois l'application lancée, SAFRAN permet d'en contrôler certains aspects programmatiquement, en particulier l'attachement et le détachement de politiques aux composants. Toutes ces fonctionnalités sont accessibles par l'intermédiaire de méthodes statiques de la classe `org.obasco.safran.Safran`.

#### Chargement des politiques

Avant de pouvoir attacher une politique à un composant, il faut la charger dans le système. La méthode

```
Safran.load("nomfichier.ext")
```

utilise la propriété système appropriée en fonction de l'extension du fichier (`.policy` ou `.fscript`) pour localiser le fichier en question, puis charge son contenu, qui peut être soit de nouvelles procédures soit de nouvelles politiques. Il est ensuite possible d'obtenir une politique grâce aux méthodes

```
Safran.getPolicy(name);
Safran.getPolicies();
```

La première renvoie la politique d'adaptation dont le nom est spécifié en paramètre (ou `null` s'il n'y en a pas), et la seconde renvoie un tableau contenant toutes les politiques connues par SAFRAN à ce moment.

#### Localisation des composants

Par défaut, Fractal n'offre pas de moyen simple pour découvrir l'ensemble des composants existants dans une application. Toutes les interfaces de Fractal sont donc inutilisables si l'on ne connaît pas à l'avance l'identité d'au moins un composant. Pour simplifier la tâche du programmeur qui doit introspecter l'application et localiser les composants à adapter, SAFRAN garde une trace de tous les composants racine présents dans le système, c'est-à-dire tous ceux qui ne font pas partie d'un sous-composant. La méthode

```
Safran.getRootComponents()
```

renvoie un tableau contenant tous ces composants racines. À partir de ces points de départ, il est relativement facile de découvrir toute l'architecture de l'application avec les interfaces de Fractal. SAFRAN simplifie encore les choses en permettant d'évaluer directement une expression FPath relativement à un composant quelconque, par exemple l'une des racines, grâce à la méthode

```
Safran.navigate(start, expression);
```

Par exemple, si `root` est l'unique composant racine de l'application, il est trivial de découvrir tous ses composants adaptatifs :

```
Safran.navigate(root, "descendant-or-self::*[interface::adaptation-controller]");
```

Enfin, la dernière méthode fournie par la classe `Safran` permet de communiquer avec WildCAT ou plus précisément l'unique instance de WildCAT présente dans chaque application adaptative :

```
org.obasco.wildcat.Context ctx = Safran.getContext();
```

L'interface de programmation complète de WildCAT (cf. Section 5.4) est alors disponible.

**Console d'administration**

Afin de faciliter le contrôle à l'exécution des applications adaptatives, SAFRAN fournit une console d'administration interactive. Cette console se présente sous la forme d'un simple « shell » qui permet d'exécuter des expressions FPath, des actions FScript, et d'invoquer certaines commandes spéciales, qui correspondent aux APIs décrites ci-dessus.

Pour démarrer cette console, il suffit d'invoquer `Safran.startConsole()` à n'importe quel moment de l'exécution du programme (typiquement au démarrage). La console utilise la sortie et l'entrée standard du programme, et affiche une invite :

```
Safran >
```

L'utilisateur peut alors entrer des expressions FPath ou FScript, qui sont évaluées / exécutées, et dont la valeur de retour est ensuite affichée. La fonction FPath `roots()`, disponible uniquement dans la console, correspond à la méthode `Safran.getRootComponents()` et permet donc d'obtenir l'ensemble des composants racines. Par exemple :

```
Safran > roots()/descendant-or-self::*[stopped(.)]
#<nodeset: application, application/subcomponent1, application/subcomponent2>
```

Cette requête sous la forme d'une expression FPath affiche la liste de tous les composants qui ne sont pas encore démarrés. Il est bien sûr possible d'invoquer des actions FScript et de définir et utiliser des variables. Par exemple, pour démarrer tous ces composants :

```
Safran > stopped := roots()/descendant-or-self::*[stopped(.)];
Safran > foreach c in $stopped { start($c); }
```

Enfin, la console définit un certain nombre de commandes spéciales, qui ne correspondent pas à des procédures FScript :

- `!load` *`filename`* charge un fichier, qui peut contenir soit des définitions de procédures FScript (fichiers `.fscript`), soit des politiques d'adaptation (fichiers `.policy`). Les procédures ainsi chargées sont alors immédiatement utilisables, aussi bien avec la console que dans d'autres procédures ou dans des politiques. Une fois les politiques chargées, elles peuvent être utilisées avec les actions `attach()` et `detach()` :
  ```
  Safran > !load email.policy
  Safran > attach($c, "disconnected-mode");
  ```
- `!close` permet de quitter la console, sans terminer l'application.
- `!quit` quitte la console et termine l'application en cours (équivalent à `System.exit()`).

## 8.3 Méthodologie et critères d'évaluation des applications

Pour chacune des applications exemples décrites ci-après, nous utilisons la méthodologie d'évaluation suivante :

1. Dans un premier temps, nous présentons l'application initiale, *non-adaptative*, en décrivant ses fonctionnalités et son architecture. Il s'agit dans tous les cas d'applications conçues en utilisant le modèle de composants Fractal « standard », c'est-à-dire sans les extensions introduites dans ce document. Ces applications sont donc *adaptables*, au sens où la dynamicité de Fractal nous permet de les reconfigurer, mais pas *adaptatives*, puisqu'elles n'incluent pas le code nécessaire la mise en œuvre du processus d'adaptation.
2. Nous identifions ensuite un certain nombre de *scénarios d'adaptation* pertinents pour chaque application. Chacun de ces scénarios représente une adaptation particulière de l'application initiale en fonction des évolutions de son contexte d'exécution. Nous justifions pourquoi ces différents scénarios ne sont pas programmés directement dans l'application initiale : soit ils ne pouvaient raisonnablement pas être anticipés au moment du développement de l'application, soit ils pouvaient l'être

mais leur programmation « en dur » aurait nuit à la qualité du code résultant (compréhensibilité, modularité, réutilisabilité).

3. Nous montrons alors comment chacun de ces scénarios peut être implémenté simplement par des politiques d'adaptation SAFRAN. Nous donnons le code de chacune de ces politiques, indiquons à quel(s) composants elles sont destinées et expliquons leur principe de fonctionnement et les fonctionnalités de SAFRAN utilisées. En associant ces politiques d'adaptation SAFRAN aux composants de l'application initiale adaptable, nous obtenons une *application adaptative*.
4. Enfin, nous évaluons cette application adaptative, selon deux points de vue : le niveau d'adaptabilité de l'application, et la facilité (ou complexité) du processus de développement qui a conduit à ce résultat.

   Le premier point de vue, qui évalue la qualité des adaptations dont est capable l'application, indique dans quelle mesure la nouvelle application est « meilleure » que sa version initiale. Par exemple, si l'objectif d'un scénario particulier était d'améliorer les performances de l'application dans certaines circonstances, la qualité de l'adaptation correspond aux gains effectifs de la version adaptative de l'application. Cette première évaluation est intéressante car elle permet de montrer que les adaptations rendues possibles par SAFRAN sont effectives ; il s'agit donc d'évaluer le *pouvoir d'expression* de SAFRAN.

   Cependant, ce point de vue n'est pas suffisant car chaque scénario pris individuellement aurait pû être implémenté de façon *ad hoc*. En ne mesurant que le résultat final, ce critère ne prend pas en compte l'objectif principal de SAFRAN, à savoir rendre plus simple et efficace *le développement* des applications adaptatives lui-même. Le second point de vue s'intéresse donc au processus de développement qui a conduit à cette solution : pour un résultat donné (une application adaptative), en quoi est-il plus intéressant d'utiliser SAFRAN que des techniques *ad hoc*, développées au cas par cas pour chaque scénario ?

Les critères d'évaluation sont ceux que nous avions identifiés initialement dans le chapitre 1. Nous les avions présentés alors selon un découpage « fonctionnel » correspondant aux différentes parties d'une application adaptative : mécanismes de reconfiguration, sensibilité au contexte et stratégies d'adaptation. Dans le cadre de l'évaluation globale du système qui est l'un des objectifs de ce chapitre, il est plus pertinent de regrouper ces mêmes critères selon qu'ils s'appliquent plus à l'application adaptative finale (premier point de vue ci-dessus) ou bien au processus de développement (second point de vue) :

- Le critère de *performance*, qui s'appliquait aux mécanismes de reconfiguration et à la sensibilité au contexte permettent d'évaluer la qualité du résultat.
- Les *garanties* offertes par les mécanismes de reconfiguration (FScript dans notre cas) contribuent essentiellement à la qualité de l'application finale, en assurant que les adaptations effectuées ne risquent pas de la rendre inutilisable.
- La *modularité*, l'*ouverture* et la *transparence* de ces même mécanismes de reconfiguration s'appliquent au contraire au processus de développement de l'application adaptative, qu'elles simplifient en découplant le code de reconfiguration du code métier.
- La *précision* et la *richesse* des informations contextuelles s'appliquent au résultat, alors que la *généralité* du mécanisme de sensibilité au contexte s'applique au processus.
- Enfin, la *dynamicité*, la *généricité* et la *réutilisabilité* des politiques d'adaptation simplifient le développement, indépendemment de la qualité du résultat. Le cas de l'*analysabilité* de ces mêmes politiques est plus ambiguë, car cette propriété permet à la fois de garantir un bon fonctionnement des politiques à l'exécution, et simplifie aussi leur développement grâce à une structuration forte qui facilite leur compréhension.

Nous avons choisi d'illustrer l'utilisation de SAFRAN sur deux applications exemples. La première (Section 8.4) est un lecteur de courrier électronique, exemple typique d'un logiciel graphique destiné aux utilisateurs finaux. La seconde (Section 8.5) est un petit serveur web (HTTP[1]), plus représentatif du type de logiciels que l'on peut trouver dans un système d'information d'entreprise. Le plan de chaque section reprend les différentes étapes de la méthodologie décrite ci-dessus : présentation de l'application initiale,

[1] Hyper-Text Transfer Protocol

puis des scénarios, et enfin évaluation. Nous concluons le chapitre par une évaluation plus globale des avantages et limitations actuelles de SAFRAN.

## 8.4 Exemple 1 : lecteur de courrier électronique

### 8.4.1 Présentation de l'application

La première application que nous avons choisie est un lecteur de courrier électronique graphique. Cette application, développée à base de composants Fractal, offre uniquement les fonctionnalités de base de ce type de logiciels : téléchargement des messages depuis un serveur distant, consultation de ces messages, et envoi de courriers. L'interface graphique est très classique : une fenêtre principale divisée en trois partie (dossiers, liste de messages et aperçu du message sélectionné) et une seconde fenêtre destinée à l'édition de nouveaux courriers.

L'architecture interne de l'application est représentée par la figure 8.2. Le composant Fractal qui représente l'application dans son ensemble est un composite qui contient les sous-composants suivants :

1. un composant `ui` (*User Interaction*) qui encapsule tous les composants interagissant avec l'utilisateur (fenêtres graphiques et système de notification) ;
2. un composant `store` qui représente la base de données des messages ;
3. et enfin `connexion`, qui est chargé de la communication avec le serveur de mail pour l'envoi et la réception des messages.

Nous n'entrerons pas plus ici dans les détails de chacun de ces composants. Ceux qui interviennent directement dans les scénarios d'adaptation seront décrits au moment opportun.

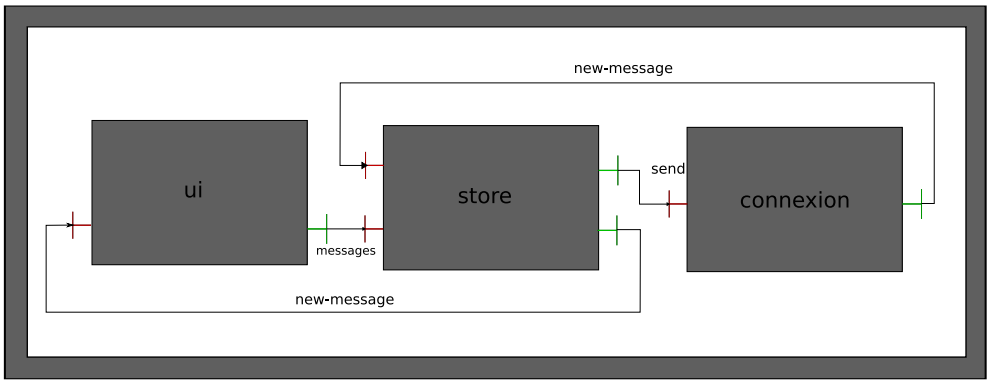


FIG. 8.2 – Architecture interne du lecteur de courrier

L'application complète est distribuée sous la forme d'un fichier `.jar` contenant toutes les classes Java et les ressources nécessaires, ce qui inclut en particulier les fichiers `.fractal` qui décrivent son architecture. Notons que cette application a été conçue sans tenir compte ni de SAFRAN ni des extensions de Fractal que nous avons introduites dans ce document. Nous nous interdisons par ailleurs de modifier son code, que l'on suppose disponible uniquement sous forme compilée (fichiers `.class`).

### 8.4.2 Envoi de courriers en mode déconnecté

La version initiale de l'application a été conçue pour être utilisée dans un environnement contrôlé, dans lequel a connexion réseau nécessaire à la réception et à l'envoi de messages est toujours disponible. Elle ne possède pas de « mode déconnecté » dans lequel les envois de messages sont différés lorsque la connexion n'est pas disponible. L'arrivée du nomadisme et des connexions sans fil moins fiables rend cette décision initiale caduque, et l'application doit donc être adaptée aux nouveaux modes d'utilisation. Notre

premier scénario va donc consister à développer une politique d'adaptation qui implémente un mode déconnecté activé lorsque le réseau n'est plus disponible.

**Informations contextuelles.** La seule information contextuelle dont nous avons besoin pour ce scénario concerne la présence ou l'absence d'une connexion réseau. Nous utilisons pour cela le domaine contextuel `sys ://` de WildCAT, qui représente les ressources système de la machine hôte. En particulier, nous nous intéressons à la ressource `sys ://network/interfaces` qui regroupe toutes les informations sur les interfaces réseau disponibles, chaque interface réseau disponible étant modélisée par une sous-ressource. Ainsi, l'interface réseau principale est représentée par la ressource `sys ://network/interfaces/eth0`, qui n'existe que si l'interface en question est active. Pour cela, la ressource `interfaces` utilise une sonde qui renvoie sous forme d'attributs l'ensemble des interfaces réseaux actives à un moment donné (en lisant le contenu du fichier spécial `proc/net/if_inet6` sous Linux). Les capacités de modélisation dynamique de WildCAT sont alors utilisées pour ajouter ou retirer dynamiquement au domaine contextuel les sous-ressources correspondant à chaque interface réseau :

```
<resource name="interfaces">
  <sensor name="nics" class="org.obasco.wildcat.sensors.NicSensor" />
  <foreach variable="itf" in="@*">
    <resource name="$itf">
      <sensor class="org.obasco.wildcat.sensors.NetConnectionSensor">
        <configuration>
          <device>$itf</device>
        </configuration>
      </sensor>
    </resource>
  </foreach>
</resource>
```

Cet extrait du fichier `system.xml` qui modélise le domaine contextuel `sys ://` fonctionne de la façon suivante :

- La sonde `nics` renvoie régulièrement un ensemble d'attributs correspondant aux interfaces réseau disponibles : le nom de l'attribut correspond au nom de l'interface, et sa valeur est simplement *vrai*.
- Lorsqu'une interface réseau est activée ou désactivée, l'attribut correspondant disparaît.
- Suivant les indications de l'élément `foreach`, si un attribut nommé `itf` apparaît WildCAT instancie une nouvelle ressource de même nom, associée à un sonde configurée spécifiquement pour observer cette interface. Cette sonde fournit des informations détaillées sur l'interface réseau, que nous n'utiliserons pas dans ce scénario.
- À l'inverse, si un attribut `itf` disparaît, la ressource correspondante est supprimée du domaine contextuel (après avoir stoppé la sonde qui lui était associée).

Encore une fois, les sondes et la modélisation décrite ici sont très génériques et une fois développées sont réutilisables facilement dans de nombreuses applications.

**Événements.** Étant données ces informations, les événements qui vont nous intéresser pour notre scénario sont donc :

- `disappears(sys ://network/interfaces/eth0)`, qui indique que la ressource `eth0` a été supprimée du contexte, c'est-à-dire que la connexion réseau vient d'être coupée.
- `appears(sys ://network/interfaces/eth0)`, qui indique au contraire que la connexion réseau est rétablie.

Notre politique d'adaptation va donc être constituée de deux règles réactives qui activent ou désactivent le mode déconnecté lorsque l'un de ces deux événements se produit.

**Communications réseau dans l'application initiale.** Dans l'architecture de l'application initiale, les communications réseau avec le serveur de mail sont encapsulées dans le composant `connexion`. C'est

donc ce composant et uniquement lui qui sera la cible de notre politique. Ce composant dispose de deux interfaces de service :

- une interface fournie `sender`, qui est utilisée pour envoyer un nouveau courrier (protocole SMTP[2]) ;
- une interface requise optionnelle `message-listener`, utilisée pour notifier la réception d'un courrier (protocole POP3[3]).

De plus, ce composant dispose d'un paramètre de configuration (interface `attributecontroller`) nommé `pollInterval` qui indique le délai entre deux requêtes POP3 auprès du serveur distant (en secondes). Si la valeur de ce paramètre est 0, alors le composant n'essaiera pas de récupérer le courrier.

**Mode déconnecté.** Afin de mettre cette application en mode déconnecté, nous devons empêcher le composant `connexion` de communiquer avec le réseau. Pour la partie « réception de messages », il suffit de modifier la valeur du paramètre `pollIntervall`, ce qui peut se faire simplement en FScript avec l'action primitive `set-value()`. En revanche, le cas de l'envoi de courrier est plus complexe. Normalement, lorsqu'une demande d'envoi de courrier est reçu par le composant sur son interface `sender`, il communique directement avec le serveur SMTP, et a donc besoin de la connexion réseau. Nous devons donc intervenir avant que le composant exécute le message reçu par `sender`. La solution est d'utiliser un méta-composant qui retarde l'exécution de ces messages jusqu'au retour de la connexion réseau. Ce méta-composant, nommé `deferred-messages` est en fait totalement générique et indépendant du lecteur de mail. Il fonctionne de la façon suivante :

1. S'il est *démarré* et qu'il reçoit une invocation de méthode destinée à un composant de base à travers son interface `message-handler`, alors il stocke toutes les informations concernant ce message (destinataire, méthode et arguments) dans une file (FIFO), et renvoie une valeur `null`[4].
2. Lorsqu'il est arrêté (`LifeCycleController.stopFc()`), le méta-composant « rejoue » les messages qu'il avait mémorisé, dans leur ordre d'arrivée initiale, et les retire de sa file.

L'association de ce méta-composant au composant de base `connexion` va donc faire en sorte que tous les envois de courriers soient retardés, jusqu'au moment où le méta-composant sera stoppé et déconnecté.

**Politique d'adaptation.** Nous avons maintenant tous les éléments nécessaires pour écrire la politique d'adaptation destinée au composant `connexion` :

```
policy disconnected-mode = {
  rule {
    when disappears(sys://network/interfaces/eth0)
    do {
      if ($target/sibling::deferred-messages) then {
        set-meta($target, $target/sibling::deferred-messages);
      } else {
        meta := new("deferred-messages");
        // Intègre le nouveau composant dans l'application
        foreach p in $target/parent::* {
          add($p, $meta);
        }
        set-meta($target, $meta);
      }
      start($target/meta::*);
      set-value($target/@pollInterval, 0);
    }
  }
```

[2]Simple Mail Transport Protocol

[3]Post-Office Protocol v3

[4]Puisqu'il est générique, ce méta-composant ne peut pas renvoyer d'autre valeur plus spécifique. Il ne fonctionnera donc que si les méthodes réifiées renvoient `void` (ce qui est la cas dans notre exemple) ou si la valeur renvoyée n'est pas importante pour le composant appelant. Une version plus sophistiquée pourrait renvoyer un objet futur [Baude et al., 2000].

```
  rule {
    when appears(sys://network/interfaces/eth0)
    do {
      if ($target/meta::*) {
         stop($target/meta::*);
         unset-meta($target);
      }
      set-value($target/@pollInterval, 5*60);
    }
  }
}
```

Cette politique est constituée de deux règles. La première est déclenchée lorsque la connexion réseau disparaît, et active le mode déconnecté. Pour cela, elle associe au composant cible `$target` le méta-composant décrit plus haut, en le créant et en l'intégrant dans l'application s'il n'est pas déjà présent, puis désactive la réception de nouveaux courriers en mettant à zéro la valeur du paramètre `pollIntervall`. La seconde règle, déclenchée lorsque la connexion est rétablie, stoppe le méta-composant (ce qui déclenche l'envoi des courriers en attente), le déconnecte, et rétablie la réception de nouveaux courriers[5].

**Déploiement.** Une fois cette politique développée, le fichier correspondant doit être chargé dans SAFRAN pendant l'exécution de l'application. Le plus simple est d'utiliser la console d'administration :

```
> !load disconnected-mode.policy
```

Cette commande est équivalente à `Safran.load("disconnected-mode.policy")`, et recherche le fichier en question dans l'un des répertoires désignés par la propriété système `org.obasco.safran.policies.path`. Il ne reste alors plus qu'à attacher une instance de cette politique au composant à adapter, en l'occurrence le composant `connexion` :

```
> attach($root/child::connexion, "disconnected-mode");
```

où `$root` désigne le composant racine de l'application.

À partir de cet instant, l'application s'adaptera automatiquement à la présence ou non d'une connexion réseau, en activant le mode déconnecté de façon transparente lorsque c'est nécessaire.

### 8.4.3 Mode de notification adapté au contexte d'utilisation

Lorsqu'un nouveau courrier est reçu par le composant `connexion` (seul à être en communication directe avec le serveur de courriers), ce dernier l'envoie à la base de donnée des messages (`store`), qui le stocke et prévient à son tour le composant `ui` pour qu'il puisse notifier l'utilisateur. Le composant `ui` dispose de deux méthodes de notification :

- une notification visuelle simple et non-intrusive sous la forme d'une icône dans la barre de tâches du bureau de l'utilisateur ;
- une notification sonore, sous la forme d'un fichier audio du type « Vous avez un message ».

L'application initiale permet à l'utilisateur de choisir l'une de ces méthodes dans son panneau de configuration. Si cette possibilité de choix est une bonne chose pour l'utilisateur en ce qu'elle lui permet d'adapter le comportement de l'application à ses préférences, la façon dont elle se présente pose problème. En effet, la méthode de notification la plus appropriée à un moment donné dépend du contexte d'utilisation de l'application, et un choix statique comme celui proposé ne conviendra pas à toutes les situations :

- Si l'utilisateur est absent de son bureau, une notification sonore est inutile, et peut même être gênante pour les personnes proches non concernées. De plus, une telle notification étant par nature ponctuelle, l'utilisateur ne saura pas en revenant à son poste s'il a reçu du courrier.

[5] Une limitation actuelle de SAFRAN est que les politiques n'ont pas d'état propre. Dans cet exemple, cela signifie qu'il n'est pas possible de mémoriser l'ancienne valeur du paramètre `pollIntervall`.

– En revanche, la notification sonore être plus utile si l'utilisateur est présent car il peut la percevoir sans interrompre son activité, même s'il ne travaille pas directement sur son ordinateur (dans ce cas, une notification visuelle passerait inaperçue).

Notre second scénario aura donc pour objectif de rendre *adaptative* la sélection du mode de notification utilisé. Ainsi, l'application utilisera toujours le mode le plus approprié au contexte actuel et en particulier à l'activité de l'utilisateur, sans intervention de la part de ce dernier.

La partie de l'application initiale qui correspond à la fonction de notification est représentée par la figure suivante 8.3. La demande de notification est reçue par l'interface `notification` du composant `ui`, qui la redirige vers un de ses sous-composants. Chaque mode de notification est implémenté par un composant Fractal distinct fournissant cette même interface. Le composant utilisé est celui connecté à l'interface interne du composite `ui`. Dans l'application initiale, le choix explicite de l'utilisateur par l'intermédiaire du panneau de configuration se traduit par la connexion de l'un ou l'autre des composants disponibles.

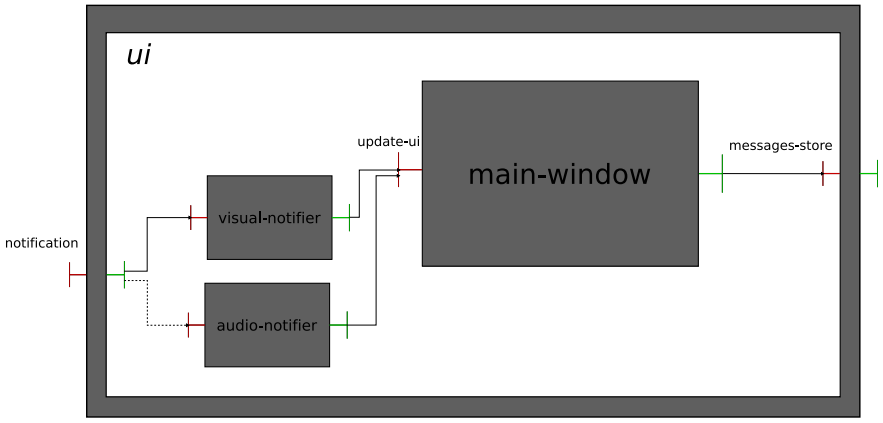


FIG. 8.3 – Système de notification dans l'application initiale.

L'implémentation du scénario d'adaptation décrit ci-dessus avec SAFRAN se fait très simplement en créant une politique d'adaptation qui sera associée au composant `ui`. Les différentes règles qui constituent cette politique détectent les changements significatifs dans l'activité de l'utilisateur, et reconfigurent le composant cible `ui` afin d'activer le mode de notification appropriée à chaque situation.

**Informations contextuelles.** Puisque l'activité de l'utilisateur fait partie du contexte d'exécution, elle est réifiée par WildCAT, et ses changements se traduisent par l'occurrence d'événements exogènes auxquels les politiques d'adaptation peuvent réagir. Concrètement, WildCAT est configuré pour fournir un domaine contextuel nommé `user`, qui regroupe et organise toutes les informations concernant l'état et l'activité de l'utilisateur. Nous n'entrerons pas dans les détails techniques de l'implémentation des sondes nécessaires. Notons cependant que de telles sondes, et plus globalement le domaine contextuel `user`, ont un intérêt beaucoup plus général que notre scénario spécifique. Bien que la modélisation du domaine et le développement des sondes nécessaires soit relativement coûteux, ce travail n'est à effectuer qu'une seule fois et peut ensuite être réutilisé par de nombreux scénarios dans des applications très diverses[6]. Concrètement, les éléments du domaine `user` qui nous intéressent pour notre scénario sont les suivants :

– `user ://location/current/logical@room` est le nom de la salle dans laquelle est actuellement l'utilisateur, par exemple B-206. Ce type d'information peut être connu par exemple par l'utilisation de badges, comme dans le projet Active Badge d'Olivetti [Want et al., 1992], d'ailleurs considéré par beaucoup comme la première application « *context-aware* », i.e. sensible au contexte.
– `user ://location/office@room` est le nom du bureau de l'utilisateur. Il s'agit d'une information statique facilement récupérable par exemple dans un annuaire de type LDAP.

[6]Faute de temps et de ressources, l'implémentation actuelle de ce domaine est une simulation.

– `user ://activity@working` est un booléen qui indique si l'utilisateur est actuellement en train de travailler. En première approximation, on considère que l'utilisateur est en train de travailler si l'on détecte une activité sur son ordinateur (clavier, souris) dans la dernière minute.

**Politique.** La politique d'adaptation correspondant au scénario est la suivante :

```
action set-notification-mode(target, mode) = {
  if ($target/binding::notification != $mode) {
    unbind($target/internal-interface::notification);
    bind($target/internal-interface::notification, $mode);
  }
}

policy adaptive-nofitication = {
rule {
  when realized(user://location/current/logical@room != user://profile/office@room)
  do {
    if (not($target/child::icon) {
      notifier =  new("example.mail.VisualNotifier");
      set-name($notifier, "icon");
      add($target, notifier);
      start($notifier);
    }
    set-notification-mode($target, $target/child::icon/interface::notifier);
  }
}

rule {
  when realized(user://location/current/logical@room == user://profile/office@room
                and user://activity@working)
  if (not(user://profile/disabilities@hearing_impaired))
  do {
    if (not($target/child::sound) {
      notifier =  new("example.mail.AudioNotifier");
      set-name($notifier, "sound");
      add($target, notifier);
      start($notifier);
    }
    set-value($notifier/@volume = 0.3);
    set-notification-mode($target, $target/child::sound/interface::notifier);
  }
}
}
```

Cette politique est constituée de deux règles, qui utilisent une action FScript définie en dehors de la politique elle-même (mais dans le même fichier). C'est cette action `set-notification-mode` qui effectue la reconfiguration elle-même, en s'assurant que le mode de notification demandé (`mode`, une interface) est bien connecté. La première règle de la politique détecte le départ de l'utilisateur de son bureau, grâce à un descripteur d'événement exogène utilisant deux des attributs WildCAT décrits plus haut. Lorsque cela se produit, la réaction correspondante est d'effecteur la connexion du mode de notification approprié, grâce à `set-notification-mode`, après s'être assuré que le composant qui implémente ce mode de notification est bien présent et prêt à être utilisé. La seconde règle est similaire, mais réagit à un événement différent, à savoir « l'utilisateur est dans son bureau et en train de travailler », et sa réaction est d'activer le mode de

notification sonore, mais uniquement si l'utilisateur n'est pas malentendant (auquel cas une notification sonore est bien sûr inutile). Pour s'assurer que l'utilisateur entendra bien la notification mais afin qu'elle ne le dérange pas trop (puisqu'il travaille), le composant de notification sonore est configuré pour utiliser un volume sonore relativement faible.

### 8.4.4 Évaluation

Concernant le niveau d'adaptabilité obtenu grâce à l'utilisation de SAFRAN, l'ajout du mode déconnecté étend le domaine de fonctionnement de l'application par rapport à la version initiale, répondant ainsi à la problématique de la variabilité du contexte d'exécution. La seconde politique d'adaptation rend l'application *plus autonome* que sa version initiale, en évitant à l'utilisateur de la reconfigurer explicitement.

Si en toute rigueur l'application adaptative est sans doute moins *performante* que sa version de base, le temps mis pour effectuer les reconfigurations est négligeable dans le cas d'une application interactive, et ce pour les deux scénarios. La *précision* et la *richesse* des informations contextuelles dépendent bien évidemment de l'implémentation des domaines contextuels utilisés (ici `user`[7] et `sys`), mais encore une fois, le coût du développement de ces domaines contextuels peut être amorti sur de nombreuses applications grâce à la *généralité* de WildCAT.

En ce qui concerne le processus de développement de l'application adaptative avec SAFRAN, par rapport à une implémentation *ad hoc*, l'*ouverture* et la *transparence* des mécanismes de SAFRAN nous ont permis d'intégrer nos deux scénarios d'adaptation dans l'application sans avoir à modifier son code, et sans même que le concepteur initial de l'application ait prévu explicitement l'existence de ces scénarios. Cependant, la mise en place de ces scénarios nécessite l'apprentissage de SAFRAN et de ses sous-systèmes (en particulier WildCAT et FScript).

La *dynamicité* de SAFRAN autorise le développement des politiques non pas « à l'aveugle » avant de déploiement de l'application, mais uniquement lorsque le besoin s'en fait ressentir, et une fois le problème bien identifié. Cette dynamicité permet aussi de développer des politiques de façon interactive et incrémentale, en les testant et en les modifiant au fur et à mesure, sans risquer de compromettre l'intégrité de l'application grâce aux *garanties* offertes par le système : si une action de reconfiguration est mal programmée et génère une erreur, FScript la détectera et garantit un retour à la configuration initiale de l'application.

Les deux politiques sont *modulaires*, en ce qu'elles ne font référence qu'aux composants qu'elle manipulent explicitement, et sont donc indépendantes du reste de l'application. Si les mêmes composants de notification ou de communication réseau étaient utilisés dans une autre application, par exemple de messagerie instantanée, les deux politiques pourraient être *réutilisées* telles quelles.

Enfin, si l'application vient à évoluer, par exemple avec le développement de nouveaux modes de notification (envoi de SMS...), les politiques d'adaptation peuvent être facilement modifiées pour prendre en compte ces nouveaux composants, encore une fois sans modifier le code de l'application.

## 8.5 Exemple 2 : serveur web

### 8.5.1 Présentation de l'application

La seconde application exemple que nous avons choisie pour illustrer l'utilisation de SAFRAN est un petit serveur web nommé Comanche, développé par É. Bruneton dans le cadre d'un tutoriel sur la programmation avec Fractal [Bruneton, 2004]. Contrairement au lecteur de courriers de la section précédente, cette application de type serveur n'interagit pas directement avec un utilisateur. Cependant, le besoin de performances implique qu'elle doit être capable de s'adapter aux caractéristiques de son hôte et au type de charge qu'elle subit. Dans cet exemple, le contexte significatif pour l'adaptation sera donc celui des ressources matérielles et logicielles plutôt que les caractéristiques de l'utilisateur final.

[7] Actuellement une simple simulation.

### 8.5.2 Amélioration des performances par ajout d'un cache adaptable

Comanche, se voulant extrêmement simple, n'intègre pas de mécanisme pour mettre en cache le contenu des fichiers qu'il lit. Afin d'améliorer ses performances, nous décidons donc d'introduire un nouveau composant adaptatif qui ajoute cette fonctionnalité.

Les performances du composant cache dépendent essentiellement de la quantité de mémoire qu'il est autorisé à utiliser pour stocker le contenu des fichiers lus précédemment. Si cette quantité est trop faible, le système ne tirera pas pleinement partie de la présence d'un cache, et si elle est trop importante, les performances risquent d'être encore moins bonnes. En effet, si le cache utilise trop de mémoire le système d'exploitation devra avoir recours à de la mémoire virtuelle sur disque, et donc beaucoup plus lente (phénomène de « *trashing* »). La quantité de mémoire allouée au composant cache ne peut donc pas être une constante, mais doit être déterminée en fonction de la quantité de mémoire disponible sur le système hôte. Or, cette quantité varie au cours du temps, le serveur HTTP n'étant pas le seul programme en cours d'exécution sur le système. Si par exemple un utilisateur démarre une application gourmande en mémoire alors que le cache utilise une grande partie de la mémoire système, le système d'exploitation devra déplacer une partie de la mémoire utilisée par le cache sur le disque dur, beaucoup plus lent, pour pouvoir fournir la mémoire nécessaire au nouveau programme.

Notre premier scénario d'adaptation pour cette application va donc consister à rendre adaptable la quantité de mémoire allouée au composant cache afin de garantir de bonnes performances en toutes circonstances.

**Architecture initiale.** Dans Comanche, la gestion des requêtes HTTP elles-mêmes est traitée par un composant `request-dispatcher`. Ce dernier possède une interface cliente de type « collection » qui l'autorise à avoir plusieurs interfaces clientes connectés. Lorsque ce composant reçoit une requête HTTP, il l'envoie tour-à-tour à chacun de ses clients, jusqu'à ce que l'un d'entre eux soit capable d'y répondre (motif *Chaîne de responsabilité* [Gamma et al., 1994]). La configuration initiale de Comanche, représentée sur la figure 8.4, utilise deux composants : le premier tente de lire le contenu du fichier local mentionné dans l'URL[8] de la requête, et si il échoue, le second composant (qui lui n'échoue jamais) renvoie un message d'erreur standard (404).

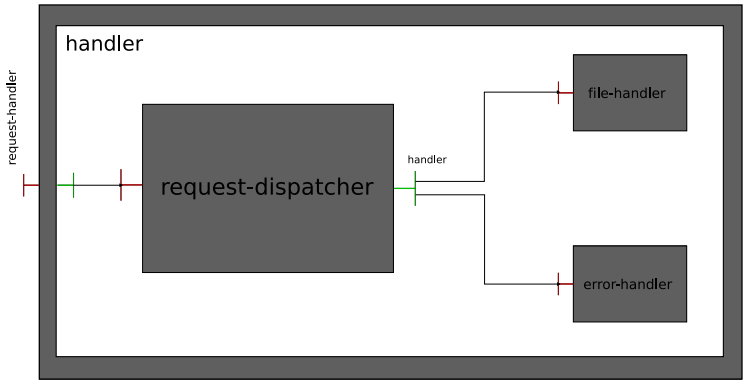


FIG. 8.4 – Architecture initiale du gestionnaire de requêtes Comanche.

**Ajout du cache.** L'introduction d'un cache de fichiers dans cette architecture se fait très simplement. Il suffit d'implémenter le composant correspondant et de l'introduire dans l'architecture ci-dessus sur le chemin du composant `file-handler`, ce qui ne nécessite qu'une modification très simple du fichier `.fractal` décrivant l'architecture de l'application. Sans rentrer dans le détail des différentes méthodes, le code métier de ce composant cache est le suivant :

[8] Uniform Resource Locator

```
public void handle(Request req) throws IOException {
    if (inCache(req.url)) {
        returnCachedVersion(req);
    } else {
        ByteArrayOutputStream data = new ByteArrayOutputStream();
        Request proxy = createProxyRequest(req, data);
        fileHandler.handle(proxy);
        if (cache(req.url, data)) {
            returnCachedVersion(req);
        } else {
            req.output.write(data.toByteArray());
        }
    }
}
```

Lorsqu'il reçoit une requête, il vérifie tout d'abord si le fichier désigné (`req.url`) n'est pas en mémoire. Si c'est le cas, il renvoie directement son contenu, sans avoir besoin d'accéder au disque. Sinon, il crée une copie de la requête initiale, modifiée pour que la réponse soit stockée dans un tampon mémoire plutôt qu'envoyée directement à travers la connexion réseau. Cette requête est ensuite envoyée au composant `file-handler`, qui remplit le tampon avec le contenu du fichier. Enfin, les données de ce tampon sont ajoutées dans le cache du composant (s'il reste assez de place), et renvoyées comme résultat de la requête. L'architecture résultante est représentée sur la figure 8.5.

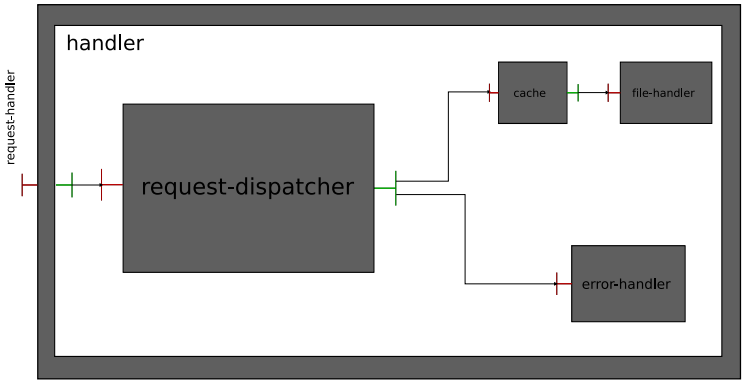


FIG. 8.5 – Introduction d'un cache de fichiers dans Comanche.

**Informations contextuelles.** Le composant cache décrit ci-dessus expose deux paramètres de configuration accessibles par son interface `attribute-controller`. Le premier, `currentSize`, est accessible en lecture seule et indique la quantité de mémoire actuellement utilisée par le cache (en kilo-octets). Le second, accessible en lecture et écriture, se nomme `maximumSize` et indique la quantité de mémoire maximale (toujours en kilo-octets) utilisable par le cache pour stocker le contenu des fichiers déjà lus. Comme nous l'avons indiqué plus haut, la valeur de ce paramètre doit être assujettie à la quantité de mémoire disponible sur le système pour offrir de bonnes performances en toutes circonstances. Notre politique d'adaptation doit donc réagir aux évolutions de cette quantité de mémoire libre. WildCAT fournit en standard une sonde observant la mémoire (en utilisant le fichier spécial `/proc/meminfo` sous Linux), et la quantité de mémoire libre est accessible par l'attribut `sys ://storage/memory@free`.

**Politique d'adaptation.** Nous avons maintenant toutes les informations nécessaires pour écrire la politique d'adaptation :

```
action disable-cache(handler) = {
  dispatcher := $handler/child::request-dispatcher;
  if (name($dispatcher/#handler/component::*) = 'cache') then {
    unbind($dispatcher/#handler);
    file-handler := $handler/child::file-handler;
    bind($dispatcher/#handler, $file-handler/interface::request-handler);
  }
}

action enable-cache(handler) = {
  dispatcher := $handler/child::request-dispatcher;
  if (name($dispatcher/#handler/component::*) != 'cache') then {
    unbind($dispatcher/#handler);
    file-handler := $handler/child::file-handler;
    cache := $handler/child::cache;
    bind($dispatcher/#handler, $cache/interface::request-handler);
    bind($cache/#handler, $file-handler/interface::request-handler);
  }
}

policy adaptive-cache = {
  rule {
    when realized(sys://storage/memory@free < 10*1024),
         not(realized(sys://storage/memory@free >= 10*1024, 10))
    do {
      to-free := 10*1024 - sys://storage/memory@free;
      size := $target/cache/@currentSize - $to-free;
      if ($size < 500) then {
        set-value($target/cache/@maximumSize, 0);
        disable-cache($target);
      } else {
        set-value($target/cache/@maximumSize, $size);
      }
    }
  }

  rule {
    when mem:changed(sys://storage/memory@free)
    if (sys://storage/memory@free >= 10*1024)
    do {
      enable-cache($target);
      size := 0.8 * ($mem.new-value + $target/cache/@currentSize);
      max := sys://storage/memory@used - $target/cache/@currentSize + $size;
      if ($max < sys://storage/memory@total - 10*1024) {
        set-value($target/cache/@maximumSize, $size);
      }
    }
  }
}
```

Ce fichier `adaptive-cache.policy` commence par définir deux actions FScript qui seront ensuite utilisées dans le corps de la politique. La première action, `disable-cache` désactive le cache en le déconnectant complètement, afin de rétablir la configuration initiale de l'application. Ainsi, lorsque le cache n'est pas utilisé, le système n'a pas à payer le coût des indirections supplémentaires. La seconde action, `enable-cache`, ré-introduit le cache dans le flot d'exécution des composants. Ces deux actions permettent donc de passer à loisir entre les deux architectures décrites précédemment.

La politique d'adaptation elle-même est constituée de deux règles. La première est déclenchée si la quantité totale de mémoire disponible sur le système passe en dessous de 10 Mo pendant au moins 10 secondes (ceci afin de ne pas régir aux conditions transitoires). Lorsque cela se produit, l'action de reconfiguration tente de libérer de la mémoire en réduisant la taille maximale du cache, ou en le désactivant complètement en dessous d'une certaine taille minimale (ici 500 Ko). La seconde règle se déclenche lorsque la quantité de mémoire libre varie[9] mais est au dessus de 10 Mo. Dans ce cas, la reconfiguration ajuste la taille allouée au cache à 80% de la quantité de mémoire utilisable, mais uniquement si cela laisse suffisamment de mémoire libre globalement.

**Déploiement.** Si le composant cache lui-même est déjà intégré dans l'application en cours d'exécution, le déploiement de la politique se fait de la même manière que les précédentes, par exemple en utilisant la console :

```
Safran> !load adaptive-cache.policy
Safran> attach($handler, "adaptive-cache");
```

Où `$handler` est le composite représenté sur les figures précédentes. Sinon, la console peut être utilisée pour introduire dynamiquement le composant cache avant de déployer la politique elle-même[10] :

```
Safran> cache := new("cache");
Safran> add($handler, $cache);
Safran> !load adaptive-cache.policy
Safran> attach($handler, "adaptive-cache");
```

À partir de cet instant, le composant cache sera activé, désactivé, et configuré dynamiquement et de façon automatique afin d'offrir les meilleures performances possibles en toutes circonstances. Notons que la nature dynamique de SAFRAN permet d'expérimenter facilement différentes politiques, par exemple pour rechercher les meilleures valeurs possibles concernant la taille maximale allouée au cache. Il suffit pour cela de retirer la politique, éditer son fichier source, puis la recharger et ré-attacher au composant :

```
Safran> detach($handler,  "adaptive-cache");
[ emacs adaptive-cachepolicy ... ]
Safran> !load adaptive-cache.policy
Safran> attach($handler, "adaptive-cache");
```

### 8.5.3 Adaptation dynamique du nombre de threads

Puisque Comanche ne supporte que les requêtes statiques correspondant à des fichiers, ses performances sont limitées essentiellement par les entrées-sorties, qui bloquent le traitement d'une requête le temps de lire les données sur le disque. Pour pallier ce problème, Comanche délègue l'ordonnancement des requêtes à un composant `scheduler`, qui peut avoir plusieurs implémentations : traitement séquentiel, un thread par requête, ou utilisation d'un *pool* de threads. Chaque implémentation a des avantages et inconvénients, mais l'utilisation d'un *pool* de threads déjà créés est le meilleur compromis entre temps de réponse et ressources utilisées. Cependant, les performances d'un tel composant d'ordonnancement dépendent de la taille pour le *pool*, c'est-à-dire du nombre de threads utilisables.

Notre dernier scénario d'adaptation va consister à modifier dynamiquement le nombre de threads alloués à l'ordonnanceur. Plus le nombre de requêtes qui peuvent effectivement être traitées en parallèle est important, plus il est intéressant d'allouer de threads pour pouvoir tirer partie de ce parallélisme. En revanche, allouer trop de threads par rapport à ce qui peut effectivement être utilisé par le système nuit aux performances, car les threads eux-mêmes sont gourmands en ressources (mémoire et changements

[9] En pratique, un tel événement n'est pas généré à chaque fois que la mémoire libre change, mais à chaque fois qu'un tel changement est détecté par WildCAT. La granularité temporelle des mesures (et donc les performances) dépendent donc de la configuration de la sonde WildCAT correspondante.

[10] On suppose que les fichiers `.class` et `.fractal` qui décrivent ce composant sont visibles par la machine virtuelle.

de contexte). Le degré de parallélisme dépend quant à lui : *(i)* du nombre total de processeurs présents sur la machine, et *(ii)* de la proportion des requêtes qui nécessitent un accès au disque, obligatoirement séquentiel.

**Informations contextuelles.** Les informations dont nous avons besoin pour écrire la politique d'adaptation sont les suivantes :

- Le nombre de processeurs présents sur la machine hôte, qui détermine le nombre maximum de threads qui peuvent réellement s'exécuter en parallèle (en supposant qu'ils ne soient pas bloqués par des entrées / sorties). Ce nombre inclut à la fois les processeurs physiques et les processeurs *virtuels* présents sur les dernières générations de puces (Hyper-Threading). Le nombre de processeurs physiques peut être obtenu de WildCAT grâce à l'expression `count(sys ://cpus/*)`, et pour chacun de ces processeurs physiques, l'attribut `@flag_ht` (pour Hyper-Threading) indique que la puce contient en fait deux processeurs. Le langage des expressions WildCAT n'est actuellement pas assez puissant pour permettre d'écrire une expression telle que `sum(sys ://cpus/*@virtual_cpus)`, où `virtual_cpus` serait un attribut synthétique valant 1 ou 2 suivant la valeur du drapeau `@flag_ht`. Actuellement, toutes les machines qui supporte l'Hyper-Threading sont homogènes, c'est-à-dire que si elles ont plusieurs processeurs physiques, ces derniers sont tous du même type. Nous utiliserons cette limitation pour définir un attribut synthétique `sys ://cpus@virtual_cpus` de la manière suivante :
  `sys://cpus@virtual_cpus = count(sys://cpus/*) * if(sys://cpus/cpu0@flag_ht, 2, 1)`
  Ce nouvel attribut indique le nombre total de processeurs disponibles sur la machine hôte.
- Le taux de réussite du composant cache, c'est-à-dire le pourcentage de requêtes qui sont traitées directement par le cache sans accéder au disque. Plus ce taux est élevé, plus le nombre de requêtes qu'il est possible de traiter en parallèle est grand. Cette valeur est accessible sous la forme d'un paramètre en lecteur seule (`hitRate`) du composant cache décrit dans le scénario précédent.

**Politique d'adaptation.** Ce scénario d'adaptation nécessite deux politiques d'adaptation, car il fait interagir des composants « éloignés » dans l'architecture de l'application (le cache et l'ordonnanceur). La politique principale, destinée au composite `frontend` qui encapsule l'ordonnanceur, est très simple :

```
policy adaptive-scheduling = {
  rule{
    when a:parameter-changed($target/child::*[contains(name(.), 'cache')]/@hitRate)
    do {
      procs := sys://cpus@virtual_cpus;
      rate := $a.new-value;
      set-value($target/scheduler/@poolSize, (2 + 4*$rate)*$procs;
    }
  }
}
```

Son unique règle détecte les événements *endogènes* correspondant aux variations du taux de réussite du cache[11]. Lorsque ce taux change, l'action de reconfiguration consiste simplement à ajuster le nombre de threads alloués à l'ordonnanceur en tenant compte à la fois du nombre de processeurs disponibles et du taux de réussite du cache. Chaque processeur se voit allouer initialement deux threads, et ce nombre est augmenté proportionnellement au taux de réussite du cache (exprimé en pourcentages). Ainsi, une machine quadri-processeur dont le cache fonctionne répond directement à 50% des requêtes se verra allouer $(2 + 4 * 0.5) * 4 = 16$ threads.

Cette première politique accède au composant cache en utilisant le chemin FPath `$target/child::*[contains(name(.), 'cache')]/@hitRate`. Or, dans l'architecture initiale, le composant `cache` fait partie du composite `handler` et non du composant cible de cette politique (`frontend`, aka `$target`),

[11] Si la machine hôte supporte l'ajout et le retrait dynamique de processeurs, il suffit de changer le descripteur d'événement pour prendre en compte ce nouveau paramètre : `parameter-changed(...) or changed(sys ://cpus@virtual_cpus)`.

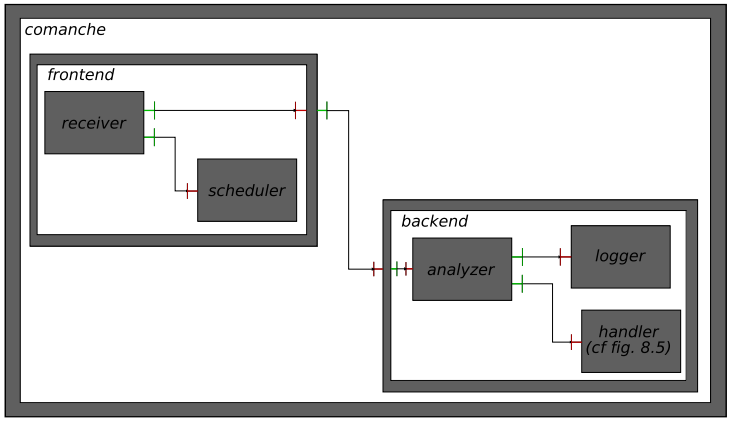


FIG. 8.6 – Architecture globale de Comanche.

comme le montre la figure 8.6. Cependant, puisque Fractal supporte le partage de composants, rien n'empêche le cache de faire aussi partie de `frontend`. C'est ce que permet d'obtenir la seconde politique de notre scénario :

```
policy adaptive-scheduling-helper = {
  rule {
    when b:binding-created($target/dispatcher/interface::*)
    if ($b.server-interface/component::*[contains(name(.), 'cache')])
    do {
      new-cache := $b.server-interface/component::*;
      frontend := $target/ancestor::comanche/frontend;
      old-cache := $frontend/cache;
      if (old-cache != new-cache) then {
        remove($frontend, $old-cache);
        add($frontend, $new-cache);
      }
    }
  }
}
```

Celle-ci est destinée au composant `handler`, tout comme la politique de gestion du cache du premier scénario. Elle réagit à l'activation du cache en détectant la création d'une connexion entre le composant `dispatcher` et un composant dont le nom contient `cache`. De cette manière, à chaque fois que la politique d'adaptation du scénario précédent ré-active le cache (action `enable-cache`), cette règle est déclenchée. Sa réaction consiste à ajouter le nouveau composant cache au composite `frontend` (après avoir éventuellement supprimé l'ancien). Le cache actif est ainsi toujours disponible localement dans `frontend`, ce qui permet le bon fonctionnement de la première politique.

Cette manipulation est rendue obligatoire par l'implémentation actuelle de SAFRAN, car pour des raisons de performances la détection des événements endogènes (dont `bindingcreated` et `parameter-changed`) est limitée en terme de portée au composant source de l'événement et à ses parents directs. Si cette limitation n'existait pas, la première politique, `adaptive-scheduling`, pourrait référencer directement le composant cache par `$target/parent::*/backend/handler/cache/@hitRate` dans son descripteur d'événement[12].

[12]Cette solution aurait cependant le désavantage de rendre la politique directement dépendante de la localisation du cache, information qui est actuellement encapsulée dans la politique `adaptive-scheduling-helper`.

**Déploiement.** Le déploiement de la politique `adaptive-scheduling` peut se faire à n'importe quel moment, mais ce n'est pas le cas de `adaptive-scheduling-helper`. En effet, cette dernière réagit à un événement endogène (l'activation du cache) qui est déclenché par une autre politique attachée au même composant. Étant données les règles de composition qui s'appliquent lorsque plusieurs politiques sont attachées à un même composant (cf. Section 7.4.4), `adaptive-scheduling-helper` doit donc être attachée *après* la politique `adaptive-cache` du premier scénario. Une fois toutes les politiques déployées, le nombre de threads alloués à l'ordonnanceur sera ajusté automatiquement au nombre de processeurs présents et au taux de réussite du cache, à chaque fois que ce taux change ou qu'un nouveau cache est activé.

### 8.5.4 Évaluation

Les deux scénarios ci-dessus permettent d'augmenter les *performances* du serveur web. Même sans SAFRAN, l'ajout d'un cache comme l'utilisation d'un ordonnanceur à base de threads auraient permis d'améliorer les performances de Comanche par rapport à sa version de base, mais les versions adaptatives de ces deux composants sont grâce à SAFRAN capable de se reconfigurer *dynamiquement* et de façon *autonome* pour éviter les problèmes de versions codées en dur (trashing, ralentissement dû aux changements de contextes lorsque trop de threads sont créés).

Les informations contextuelles nécessaires à ces deux politiques sont relativement simples et ne posent pas de problème de performances particulier (si les sondes WildCAT sont correctement configurées bien sûr). La *richesse* de WildCAT nous a permis de créer l'attribut synthétique `sys://cpus@virtual_cpus` dont nous avions besoin, mais le langage des expressions utilisables pour cela n'est pas encore assez sophistiqué. Notons cependant que grâce à l'approche *framework* utilisée par WildCAT qui en fait un système très *général*, ce langage peut être étendu sans avoir le moindre impact sur le reste de SAFRAN, ni même sur les domaines contextuels WildCAT qui n'utilisent pas l'implémentation par défaut (dont le langage en question fait partie). Le traitement homogène des événements endogènes et exogènes nous a permis d'écrire une politique qui réagit à des modifications internes à l'application aussi naturellement que les politiques précédentes qui réagissaient aux évolutions du contexte d'exécution.

Concernant le processus de développement des versions adaptatives de l'application, nous avons dû développer un nouveau composant (le cache) en plus de la politique elle-même, ce qui a *complexifié* le processus. Heureusement, les caractéristiques de Fractal, en particulier le fait que l'architecture soit définie séparément des composants eux-même (*Inversion of Control*), nous ont permis d'intégrer ce nouveau composant dans l'application sans en modifier le code source. Une autre approche aurait été d'utiliser un méta-composant pour implémenter un cache transparent qui, lui, aurait pû être *générique*. Cependant, le fonctionnement interne de Comanche[13] et l'impossibilité pour un méta-composant générique de connaître la taille des données manipulées ou la date de modification du fichier correspondant nous ont obligé à avoir programmer un composant cache *ad hoc*. Pour le second scénario, nous n'avons cependant pas eu besoin de développer du code Java spécifique, et l'intégration de la politique d'adaptation dans l'application se fait de façon tout-à-fait transparente, y compris en ce qui concerne l'interaction avec la politique du premier scénario.

La possibilité de mettre au point les politiques d'adaptation *dynamiquement* est particulièrement intéressant dans le cas de ces deux scénarios, puisqu'ils dépendent tous deux de paramètres numériques que seule l'expérimentation permet de déterminer.

## 8.6 Conclusion

Dans ce chapitre, nous avons montré comment SAFRAN s'utilise en pratique et validé notre proposition à partir de quelques exemples de scénarios d'adaptation.

[13] Les réponses aux requêtes ne sont pas envoyées comme valeur de retour facilement manipulables par un méta-composant, mais en manipulant les objets passés en paramètres de façon spécifique.

Dans un premier temps, nous avons présenté les interfaces de programmation et les outils destinés aux programmeurs d'applications adaptatives (console par exemple), et expliqué comment le modèle de développement d'applications « classique » est modifié pour prendre en compte le développement et le déploiement des politiques d'adaptation SAFRAN.

Nous avons ensuite présenté notre méthodologie d'évaluation et rappelé les critères d'évaluation que nous avions identifiés initialement. Les quatre scénarios d'adaptation répartis en deux applications différentes que nous avons présentés ensuite nous ou permis à la fois d'illustrer concrètement les concepts présentés dans le reste du document et d'évaluer notre proposition. La première application adaptative, un lecteur de courrier électronique, est typique des applications graphiques interactives destinées à l'utilisateur final, et les deux scénarios d'adaptation correspondant (mode déconnecté et mode de notification) sont tous les deux destinés à rendre l'application plus autonome et son utilisation plus transparente pour l'utilisateur. La seconde application, basée sur le serveur web Comanche, est plutôt représentative des applications serveurs que l'on trouve intégrés dans les systèmes d'informations d'entreprise. Les performances sont primordiales pour ce genre d'applications, et les deux adaptations que nous y avons intégré (cache et ordonnanceur adaptatifs) permettent justement d'augmenter les performances du serveur.

Bien que chaque scénario pris individuellement aurait pû être développé et intégré de façon *ad hoc*, la *dynamicité*, la *souplesse* et la *transparence* apportés par l'utilisation de SAFRAN rendent de tels développement beaucoup plus *simples*, ce qui était l'objectif principal de nos travaux. Par exemple, à aucun moment nous n'avons eu besoin de modifier le code source des applications initiales, même si l'un des scénario a nécessité le développement d'un nouveau composant en plus de la politique SAFRAN elle-même. De plus, le couplage lâche et dynamique entre les composants métiers et les politiques qui implémentent *l'aspect d'adaptation* permet de faire évoluer facilement les politiques, par exemple pour leur mise au point interactive (facilitée par les garanties offertes par FScript). En contrepartie, l'utilisation de SAFRAN nécessite l'apprentissage d'un nouveau langage (qui pourrait être plus simple et naturel) par les programmeurs (ou administrateurs) des applications, et certains sous-systèmes de SAFRAN souffrent actuellement de limitations en terme de pouvoir d'expression (en particulier WildCAT et les descripteurs d'événements).

# Conclusion et perspectives

## Résumé des contributions

L'objectif de cette thèse était de faciliter la construction d'applications adaptatives, c'est-à-dire capables de se reconfigurer dynamiquement pour réagir aux évolutions de leur contexte d'exécution.

Nous avons proposé pour cela une approche basée sur les principes de conception suivants :

1. considérer l'adaptation dynamique comme un *aspect*, les politiques d'adaptation étant modularisées en dehors du code métier de l'application afin de pouvoir y être intégrées dynamiquement ;
2. utiliser la programmation par composants, et plus spécifiquement le modèle Fractal, pour la construction d'applications *adaptables* sur lesquelles les politiques d'adaptation peuvent se greffer ;
3. utiliser un langage dédié pour la spécification des politiques d'adaptation afin de pouvoir garantir la consistance des reconfigurations ;
4. utiliser le paradigme des règles réactives ECA pour structurer ces politiques.

Ces différents principes ont été mis en œuvre dans le cadre de SAFRAN, une extension du modèle de composants Fractal pour le développement de composants adaptatifs qui constitue notre principale contribution. Chacun des différents éléments qui constituent SAFRAN a été conçu pour être le plus générique possible, afin d'être réutilisable en dehors de SAFRAN :

- WildCAT est un service générique, configurable et extensible pour construire des applications sensibles à leur contexte d'exécution.
- FScript est un langage permettant d'effectuer des reconfigurations dynamiques d'applications Fractal qui garantit la consistance de ces reconfigurations.
- FPath est un sous-ensemble de FScript qui permet de naviguer dans des architectures Fractal et d'introspecter leurs composants sans risquer d'effets de bords, et peut être utilisé indépendemment de FScript, comme par exemple avec notre extension de Fractal pour le support des contraintes architecturales.
- Enfin, SAFRAN lui-même est un langage dédié pour programmer des politiques d'adaptation suivant le principe des règles réactives. Ces règles sont inspirées du paradigme ECA et constituées : *(i)* d'un *descripteur d'événement*, éventuellement composite, qui indique *quand* une certaine adaptation doit avoir lieu ; *(ii)* d'une *condition* optionelle, qui sert de garde ; *(iii)* et enfin d'une *action de reconfiguration* qui adapte l'application en reconfigurant son architecture. La langage SAFRAN repose sur WildCAT pour détecter les évolutions du contexte d'exécution et sur FScript pour effectuer les reconfigurations nécessaires.

SAFRAN s'intègre dans le modèle de composants Fractal sous la forme d'une nouvelle interface de contrôle qui permet d'attacher dynamiquement des politiques d'adaptation aux composants d'une application. Étant donné que l'un des principes de base de la conception de SAFRAN est de considérer l'adaptation comme un aspect, on peut voir cette interface de contrôle comme une « spécialisation » de la notion de tisseur d'aspect dans notre cas particulier, les opérations d'attachement et de détachement de politiques correspondant au tissage et détissage d'aspect.

Enfin, nous avons montré comment ces éléments pouvaient être intégrés dans le cycle de développement des applications à base de composants afin d'obtenir des applications adaptatives. Nous avons illustré ce

modèle de développement sur quelques exemples de scénarios d'adaptation qui nous ont permis de valider notre proposition par rapport aux critères d'évaluation identifiés initialement.

# Perspectives

Bien que les principes architecturaux sous-jacents à SAFRAN nous semblent « sains » et ne devraient pas être modifiés fondamentalement, de nombreuses améliorations sont possibles. Certaines correspondent à des limitations de l'implémentation actuelle, d'autres à des extensions facilement intégrables dans le cadre actuel, et d'autres à des évolutions plus profondes. La nature modulaire de SAFRAN facilite ces modifications puisqu'il est possible de faire évoluer chacun des éléments (WildCAT, FScript, le langage SAFRAN lui-même) individuellement, chaque amélioration locale enrichissant le système global.

## Analyses statiques pour FScript et SAFRAN

La version actuelle du langage FScript ne tire pas complètement partie de sa nature de langage dédié (DSL). L'un des avantages de l'utilisation d'un DSL au pouvoir d'expression volontairement limité est de permettre d'effectuer des analyses statiques plus poussées des programmes qu'avec un langage de programmation généraliste.

Actuellement, FScript fournit un certain nombre de garanties concernant l'exécution des reconfigurations (atomicité, consistance...), mais essentiellement grâce aux techniques d'implémentation utilisées. L'intégration d'un système de types statiques pourrait permettre d'effectuer des vérifications statiques sur la validité des reconfigurations. Idéalement, étant donnée la définition d'une action de reconfiguration prenant en paramètre un composant `c`, on aimerait pouvoir *inférer* les caractéristiques de `c` qui nous garantissent le bon fonctionnement de l'action. Par exemple, si le corps de l'action invoque l'action primitive `add($c, ...)`, on en déduit que `c` doit être un composite, ce qui peut être vérifié *avant* de tenter d'exécuter l'action.

À défaut d'un système de type par inférence, une autre approche pourrait être d'utiliser le *model checking* : étant donnée l'architecture initiale de l'application et les actions de reconfigurations disponibles, l'application de ces actions peut être simulée sans avoir à lancer l'application, afin de vérifier à l'avance que toutes les séquences de reconfigurations qui peuvent être déclenchées à l'exécution se produisent sans erreur.

La vérification du comportement des politiques SAFRAN elles-mêmes – et non plus des actions FScript seules – est plus complexe. En effet, le déclenchement de leur exécution dépend d'événements potentiellement externes sur lesquels on ne sait *a priori* que peu de choses, en particulier les événement exogènes correspondant aux évolutions du contexte, et encore moins des effets secondaires des actions de reconfiguration sur ce contexte. Par exemple, le programmeur d'une politique d'adatation sait que modifier la valeur du paramètre `maximumSize` d'un composant cache va influer sur sa consommation mémoire, et donc sur la valeur de l'attribut `sys://storage/memory@free`, mais un système de vérification automatique ne peut pas connaître le lien entre ces deux éléments. Dans ces conditions, il devient impossible de vérifier par exemple qu'une politique d'adaptation ne génère pas de boucle infinie, son action de reconfiguration entrainant une modification du contexte qui re-déclenche la même action. Le mieux que l'on puisse faire est sans doute d'offrir des outils de développement aux programmeurs de politiques leur permettant de détecter les problèmes potentiels et de *simuler* le comportement des politiques, en laissant le programmeur décider *in fine* si les problèmes détectés peuvent réellement se produire en situation réelle.

## Autres opérations de reconfiguration FScript

L'ensemble des actions primitives utilisables actuellement avec FScript recouvre les opérations standards supportées par Fractal et celles ajoutées par nos propres extensions. Cependant, de nombreuses extensions de Fractal pourraient être utiles pour enrichir les possibilités d'adaptation des applications. Citons par exemple :

– La *spécialisation* de composants [Bobeff and Noyé, 2004] pourrait être utilisée pour générer automatiquement des composants optimisés pour certaines circonstances, voire plusieurs versions d'une configuration générique, chacune spécialisée pour une situation, SAFRAN se chargeant de passer d'une implémentation à l'autre lorsque les circonstances changent.
– Le *tissage d'aspects* généralistes [Pessemier et al., 2004] (par opposition à l'aspect d'adaptation supporté par SAFRAN) permettrait d'introduire ou de retirer des préoccupations – fonctionnelles ou non – dans le programme de base.
– La *sérialisation* de composants et le *transfert d'état* permettrait de nombreuses adaptations impossibles actuellement : remplacement de l'implémentation d'un composant (par exemple pour une mise-à-jour de sécurité), migration pour la répartition de charge, passivation puis réactivation des composants gourmands en mémoire mais peu utilisés...

Aucune de ces opérations de reconfiguration n'est spécifique à SAFRAN ou à la problématique de l'adaptation. Cependant, au fur et à mesure que de nouvelles extensions de Fractal seront développées, leur support dans FScript, et donc SAFRAN pourra se faire très naturellement, puisqu'il s'agit simplement d'enrichir le vocabulaire des actions primitives disponibles.

## Observation du contexte avec WildCAT

Le modèle de données utilisé par WildCAT pourrait être étendu, d'une part afin de supporter d'autres types de données (par exemple de structures ou des collections), mais aussi pour prendre en compte la nature particulière des informations contextuelles. En effet, les informations obtenues par les sondes sont par nature imprécises et incertaines. Réifier la précision et l'incertitude des mesures permettrait de prendre des décisions plus sophistiquées au niveau des politiques SAFRAN, par exemple en ignorant certaines mesures jugées pas assez fiables pour déclencher certaines adaptation.

Une autre piste pour améliorer WildCAT serait de supporter le *raisonnement temporel* sur les évolutions du contexte. Les opérateurs de composition chroniques existant actuellement au niveau des politiques d'adaptation sont assez limités et ne permettent pas d'exprimer des conditions complexes, comme par exemple « la moyenne de l'attribut `foo` n'a pas changé de plus de 5% au cours des 10 dernières minutes ». La façon la plus simple d'ajouter ce type de fonctionnalité à WildCAT serait sans doute d'étendre le langage utilisé pour définir les attributs synthétiques en permettant d'accéder à l'historique des ressources et attributs du contexte. Cette approche a l'avantage d'être transparente du point du vue du reste de SAFRAN, « la moyenne de l'attribut `foo` au cours des 10 dernières minutes » pouvant être considéré comme un attribut WildCAT comme une autre, la façon dont il est calculé et mis à jour restant encapsulé dans l'implémentation du domaine contextuel.

## Distribution

Enfin, une extension importante de SAFRAN serait de permettre l'adaptation d'applications distribuées. On peut envisager une telle extension à différents niveaux, du plus simple au plus complexe :

1. Un premièr niveau d'extension consisterait à permettre la distribution des informations contextuelles fournies par WildCAT, comme la charge des différentes machines présentes sur un réseau, ou bien la topologie de ce réseau. En réalité, une telle extension ne nécessite pas de modifier WildCAT qui, tant que *framework*, est prévu pour supporter ce type d'extension. On peut imaginer par exemple implémenter un domaine contextuel `net://` qui expose à SAFRAN des informations pertinentes sur l'état du réseau. L'implémentation d'un tel domaine pourrait reposer sur des systèmes existants, comme SNMP[14]. Du point de vue des politiques d'adaptation SAFRAN, la nature distribuée de ces informations ne serait pas visible, et une telle extension resterait donc isolée dans les « détails d'implémentation » du domaine contextuel `net://`.
2. La distribution peut aussi être intégrée au niveau des reconfigurations FScript elles-mêmes, d'une part en faisant apparaître les connexions distribuées entre composants (Fractal RMI) au niveau de

[14]Simple Network Management Protocol

FPath et FScript, et d'autre part en permettant la migration de composants. Une telle extension nécessiterait de prendre en compte la nature distribuée des transactions de reconfigurations.

3. Enfin, en dernier lieu, la distribution pourrait être prise en compte au niveau des politiques elles-mêmes, en permettant à ces politiques de raisonner de façon distribuée et aux différentes applications adaptatives distribuées de coopérer et de coordonner leurs adaptations.

# Troisième partie

# Annexes

# Annexe A

# Référence FPath & FScript

## A.1 Syntaxe de FPath

La syntaxe concrète complète des expressions FPath est définie par la grammaire suivante :

```
expression ::= orExpression
orExpression ::= andExpression ( "or" andExpression )*
andExpression ::= compExpression ( "and" compExpression )*
compExpression ::= plusExpression ( comparator plusExpression )*
comparator ::= ( "=" | "!=" | "<" | ">" | "<=" | ">=" )
plusExpression ::= multiplyExpression ( ( "+" | "-" ) multiplyExpression )*
multiplyExpression ::= unaryExpression ( ("*" | "div") unaryExpression )*
unaryExpression ::= atom | "-" atom
atom ::= literal
       | variableReference
       | absolutePath
       | paren
       | functionCall
literal ::= number | string
variableReference ::= "$" name
paren ::= "(" expression ")"
functionCall ::= name "(" ( arguments )? ")"
arguments ::= expression ("," arguments)*
absolutePath ::= variableRefence "/" locationPath
locationPath ::= locationStep ( "/" locationStep )*
locationStep ::= name "::" test ("[" expression "]")*
test ::= name | "*"
```

## A.2 Syntaxe de FScript

La syntaxe de FScript étend celle de FPath de la façon suivante :

```
definitions ::= procedureDefinition+
procedureDefinition ::= ("function" | "action") parameters "=" blockStatement
parameters ::= "(" ( name ("," name)* )? ")"
statement ::= blockStatement
           |  expression ";"
           | "if" "(" expression ")" "then" blockStatement ( "else" blockStatement )?
```

```
          | name ":=" expression ";"
          | "return" expression ";"
          | "foreach" name "in" expression blockStatement
blockStatement ::= "{" statement+ "}"
```

## A.3 Axes de navigation FPath

Les axes de base sont les suivants :

| | |
|---|---|
| **component** | Relie un nœud *interface* ou *attribut* à celui du *composant* auquel appartient l'interface ou l'attribut. |
| **interface** | Relie un nœud *composant* à chacune de ses interfaces. |
| **attribute** | Relie un nœud *composant* à chacun de ses paramètres de configuration. |
| **binding** | Relie deux *interfaces* si et seulement si les deux interfaces sont connectées. L'orientation de l'arc correspond à l'orientation de la connexion. |
| **child** | Relie deux *composants* $A$ et $B$, dans cette direction, si et seulement si $B$ est un sous-composant direct du composite $A$. |
| **parent** | Symétrique de child. Relie les *composants* $A$ et $B$ si et seulement si $B$ est un des parents (super-composants) directs de $A$. |
| **meta** | Relie un *composant* de base à son méta-*composant*, s'il existe. |

À ces axes de base sont ajoutés des axes synthétiques :

| | |
|---|---|
| **child-or-self** | Similaire à child, mais inclue aussi le composant de départ. |
| **parent-or-self** | Similaire à parent, mais inclue aussi le composant de départ. |
| **descendant** | Sélectionne tous les sous-composants *directs et indirects* du composant de départ. Correspond à la clôture récursive de l'axe child. |
| **descendant-or-self** | Similaire à descendant, mais inclue aussi le composant de départ. |
| **ancestor** | Sélectionne tous les parents *directs et indirects* du composant de départ. Correspond à la clôture récursive de l'axe parent. |
| **ancestor-or-self** | Similaire à ancestor, mais inclue aussi le composant de départ. |
| **sibling** | Sélectionne tous les composants qui se trouvent « au même niveau » que le composant de départ, c'est-à-dire qui sont sous-composants directs d'au moins un des parents directs du composant de départ. Il s'agit en fait d'un raccourci syntaxique pour `parent::*/child::*[.!=$c]` (où `$c` représente le composant de départ). |
| **sibling-or-self** | Similaire à sibling, mais inclue aussi le composant de départ. |
| **external-interface** | Variante de l'axe interface qui ne sélectionne que les interfaces externes. |
| **internal-interface** | Variante de l'axe interface qui ne sélectionne que les interfaces internes. |

## A.4 Fonctions standards FPath

FPath fournit un certain nombre de fonctions prédéfinies, qui correspondent aux méthodes d'introspection de Fractal.

### A.4.1 Nœuds attributs

Les fonctions prédéfinies permettant d'introspecter les attributs de configuration sont les suivantes :

– `readable(attr)` $\longrightarrow$ *Boolean*
  Renvoie *vrai* si l'attribut de configuration désigné par `attr` est accessible en lecture. Si cet attribut s'apelle `foo`, cela correspond à la présence d'une méthode `getFoo()` (ou éventuellement `isFoo()` si l'attribut est booléen) dans l'interface `attribute-controller` du composant.

– `writeable(attr)` ⟶ *Boolean*
Renvoie *vrai* si l'attribut de configuration désigné par `attr` est accessible en écriture. Si cet attribut s'apelle `foo`, cela correspond à la présence d'une méthode `setFoo()` dans l'interface `attribute-controller` du composant.
– `name(attr)` ⟶ *String*
Renvoie le nom de l'attribut (`"foo"` dans l'exemple précédent).
– `value(attr)` ⟶ $String \cup Number \cup Boolean$
Renvoie la valeur courante de l'attribut. Cette fonction est invoquée automatiquement lorsqu'une expression qui désigne un attribut est utilisée dans un contexte qui nécessite une valeur.

### A.4.2 Nœuds interfaces

Les fonctions prédéfinies qui permettent d'introspecter les interfaces Fractal sont les suivantes :
– `name(itf)` ⟶ *String*
Renvoie le nom de l'interface. Correspond à `Interface.getFcItfName()`.
– `owner(itf)` ⟶ *ComponentNode*
Renvoie le nœud FPath qui désigne le composant auquel appartient cette interface. Correspond à `Interface.getFctfOwner()`.
– `internal(itf)` ⟶ *Boolean*
Renvoie *vrai* si `itf` est une interface interne. Correspond à `Interface.isFcInternalItf()`.
– `external(itf)` ⟶ *Boolean*
Renvoie *vrai* si `itf` est une interface externe. Inverse de `internal(itf)`.
– `signature(itf)` ⟶ *String*
Renvoie la signatue de l'interface, c'est-à-dire le nom qualifié de l'interface Java correspondante (par exemple `"java.lang.Runnable"`). Correspond à `InterfaceType.getFcItfSignature()`.
– `client(itf)` ⟶ *Boolean*
Renvoie *vrai* si `itf` est une interface client, i.e. requise. Correspond à `InterfaceType.isFcClient-Itf()`.
– `server(itf)` ⟶ *Boolean*
Renvoie *vrai* si `itf` est une interface serveur, i.e. fournie. Inverse de `client(itf)`.
– `optional(itf)` ⟶ *Boolean*
Renvoie vrai si `itf` est une interface optionelle. Correspond à `InterfaceType.isFcOptionalItf()`.
– `mandatory(itf)` ⟶ *Boolean*
Renvoie vrai si `itf` est une interface obligatoire. Inverse de `optional(itf)`.
– `collection(itf)` ⟶ *Boolean*
Renvoie vrai si `itf` est une interface qui supporte les connexions multiples. Correspond à `Interface-Type.isFcCollectionItf()`.
– `single(itf)` ⟶ *Boolean*
Renvoie vrai si `itf` est une interface qui ne supporte pas les connexions multiples. Inverse de `collection(itf)`.

### A.4.3 Nœuds composants

Enfin, les fonctions prédéfinies permettant d'introspecter les nœuds composants sont :
– `name(comp)` ⟶ *String*
Renvoie le nom du composant, ou une chaîne vide s'il n'en a pas. Correspond à `NameController.get-FcName()`.
– `state(comp)` ⟶ *String*
Renvoie l'état actuel du composant sous forme d'une chaîne (`"STARTED"` ou `"STOPPED"`), ou une chaîne vide si le composant n'a pas de cycle de vie explicite. Correspond à `LifeCycleController.get-FcState()`.
– `started(comp)` ⟶ *Boolean*

Renvoie *vrai* si le composant est actuellement démarré (état `"STARTED"`).
- `stopped(`*`comp`*`)` $\longrightarrow$ *Boolean*
  Renvoie *vrai* si le composant est actuellement stoppé (état `"STOPPED"`).

## A.5 Actions standards FScript

Les actions primitives (prédéfinies) fournies par FScript pour manipuler les composants Fractal correspondent aux méthodes des différents contrôleurs Fractal (standards ou ajoutés par nous) qui modifient l'état ou l'architecture des composants.

- `set-value(attribute, value)`
  Modifie la valeur du paramètre de configuration désigné par `attribute`.
- `set-name(component, name)`
  Modifie le nom du composant, si celui-ci possède l'interface `name-controller`. Correspond à `Name-Controller.setFcName()`.
- `add(composite, sub-component)`
  Ajoute un nouveau sous-composant à `composite`. Correspond à `ContentController.addFcSub-component()`.
- `remove(composite, sub-component)`
  Retire un sous-composant d'un composite. Correspond à `ContentController.removeFcSubcomponent()`.
- `bind(client-itf, server-itf)`
  Connecte deux interfaces, qui doivent être compatibles. Correspond à `BindingController.bindFc()`.
- `unbind(client-itf)`
  Supprime la connexion sortante de `client-itf`. Correspond à `BindingController.unbindFc()`.
- `start(component)`
  Démarre le composant. Correspond à `LifeCycleController.startFc()`.
- `stop(component)`
  Stopper le composant. Correspond à `LifeCycleController.stopFc()`.
- `set-meta(base-component, meta-component)`
  Change le méta-composant associé à un composant de base. Correspond à `MetalinkController.setFc-Meta()`.
- `unset-meta(base-component)`
  Retire le méta-composant associé à un composant de base. Correspond à `MetalinkController.setFc-Meta(null)`.
- `attach(component, policy)`
  Attache une nouvelle politique d'adaptation (désignée par son nom) au composant. Correspond à `AdaptationController.attachFcPolicy()`.
- `detach(component, policy)`
  Détache une politique d'adaptation (désignée par son nom) du composant. Correspond à `Adaptation-Controller.detachFcPolicy()`.
- `create(template-name)`
  Crée un nouveau composant et le renvoie. `template-name` doit être le nom du fichier `.fractal` qui contient la définition du composant à instancier.

# Bibliographie


Gul Agha. Actors : A model of concurrent computation in distributed systems. Technical Report AITR-844, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, June 1985.

Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava : Connecting software architecture to implementation. In *International Conference on Software Engineering, ICSE 2002*, Orlando, Florida, USA, May 2002.

João Paulo Andrade Almeida. Dynamic reconfiguration of object-middleware-based distributed systems. Thesis for a master of science degree in telematics, University of Twente, Enschede, The Netherlands, June 2001.

Noriki Amano and Takuo Watanabe. An approach for constructing dynamically adaptable component-based software systems using LEAD++. In *OOPSLA'99 International Workshop on Object Oriented Reflection and Software Engineering (OORaSE'99)*, pages 1–16, November 1999.

Sten Amundsen, Ketil Lund, Frank Eliassen, and Richard Staehli. QuA : Platform-managed QoS for component architectures. In *Proceedings of NIK 2004*, November 2004.

Anders Andersen, Gordon S. Blair, and Frank Eliassen. OOPP : A reflective component-based middleware. In *NIK 2000*, Bodø, Norway, November 2000.

Françoise André and Maria-Teresa Segarra. A generic approach to satisfy adaptability needs in mobile environments. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences (HICSS33)*, 2000.

Roland Balter, Luc Bellissard, Fabienne Boyer, Michel Riveill, and Jean-Yves Vion-Dury. Architecturing and configuring distributed applications with Olan. In *Proceedings Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, 1998.

Francoise Baude, Denis Caromel, Fabrice Huet, and Julien Vayssiere. Communicating mobile active objects in java. In *Proceedings of HPCN Europe 2000*, volume 1823 of *LNCS*, pages 633–643. Springer-Verlag, May 2000. URL `http://www-sop.inria.fr/oasis/Julien.Vayssiere/publications/18230633.pdf`.

Gordon S. Blair, Lynne Blair, Valérie Issarny, Petr Tuma, and Apostolos Zarras. The role of software architecture in constraining adaptation in component-based middleware platforms. In *Middleware 2000*, pages 164–184, 2000.

Gordon S. Blair and Geoff Coulson. The case for reflective middleware. Technical report, Distributed Multimedia Research Group, Department of Computing, Lancaster University, 1997.

Gordon S. Blair and Jean-Bernard Stefani. *Open Distributed Processing and Multimedia*. Addison Wesley Longman Ltd, 1998. ISBN 0-201-17794-3.

Gustavo Bobeff and Jacques Noyé. Component specialization. In *PEPM'04 : Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 39–50, Verona, Italy, 2004. ACM Press. ISBN 1-58113-835-0.

Philippe Boinot. *Une approche déclarative de la flexibilité du logiciel*. Phd thesis, Université de Rennes 1, June 2002.

Philippe Boinot, Renaud Marlet, Jacques Noyé, Gilles Muller, and Charles Consel. A declarative approach for designing and developing adaptive components. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 111–119. IEEE Computer Society, September 2000.

Noury M. N. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. *Technique et science informatiques*, 20(4/2001), 2001.

Jean-Pierre Briot and Rachid Guerraoui. Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances. *Techniques et Sciences Informatiques (TSI)*, June 1996.

P. J. Brown. Triggering information by context. *Personal Technologies*, 2(1) :1–9, September 1998. ISSN 0949-2054. URL `http://www.cs.ukc.ac.uk/pubs/1998/591`.

Éric Bruneton. Developing with fractal. `http ://fractal.objectweb.org/tutorial/index.html`, March 2004.

Éric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. An open component model and its support in java. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt Wallnau, editors, *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE 2004)*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22, Edinburgh, Scotland, may 2004. Springer-Verlag. ISBN 3-540-21998-6.

Éric Bruneton, Thierry Coupaye, and Jean-Bernard Stéfani. The fractal component model. Technical report, The ObjectWeb Consortium, September 2003. version 2.0.

Bill Burke. JBoss AOP (Aspect-Oriented Programming). Web site. `http ://www.jboss.org/developers/projects/jboss/aop`.

Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. CARISMA : Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29(10) :929–945, October 2003.

W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural reflection : Bridging the gap between a running system and its architectural specification. In *Proc. Reengineering Forum '98*, 1998.

Sharma Chakravarthy and Deepak Mishra. Snoop : An expressive event specification language for active databases. *Data Knowledge Engineering*, 14(1) :1–26, 1994.

Djalel Cherfour and François André. ACEEL : modèle de composants auto-adaptatifs. In *Actes des Journées Composants 2002 – Systèmes à Composants Adaptables et Extensibles*, Grenoble, France, October 2002.

Djalel Cherfour and François André. Développement d'applications en environnements mobiles à l'aide du modèle de composant adaptatif ACEEL. In *LMO 2003*, Vannes, February 2003a. Hermès.

Djalel Cherfour and Françoise André. Auto-adaptation de composants ACEEL coopérants. In *CFSE'3, Conférence Française sur les Systèmes d'Exploitation*, La Colle sur Loup, France, October 2003b.

Shigeru Chiba. A metaobject protocol for c++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, pages 285–299. ACM Press, October 1995.

Pierre Cointe, Jacques Noyé, Rémi Douence, Thomas Ledoux, Jean-Marc Menaud, Gilles Muller, and Mario Südholt. Programmation post-objets : des langages d'aspects aux langages de composants. *RSTI L'Objet*, 10(4), 2004. URL `http://www.lip6.fr/colloque-JFP`. À paraître.

Christin Collet. Bases de données actives : des systèmes relationnels aux systèmes à objets. Mémoire pour l'obtention du diplôme d'Habilitation à diriger des recherches RR 965-I-LSR 4, LSR-IMAG, Grenoble, France, October 1996.

Charles Consel and Renaud Marlet. Architecturing software using a methodology for language development. In *Proceedings of the 10th International Symposium on Programming Languages, Implementations, Logics and Programs PLILP/ALP '98*, Pisa, Italy, September 1998. Invited paper.

Jonathan E. Cook and Jeffrey A. Dage. Highly reliable upgrading of components. In *Proceedings of International Conference on Software Engineering (ICSE'99)*, pages 203–212, 1999.

M. Scott Corson, Joseph P. Macker, and Gregory H. Cirincione. Internet-based mobile ad hoc networking. *IEEE Internet Computing*, 3(4) :63–70, July 1999.

Simon Courtenage. Specifying and detecting composite events in content-based publish/subscribe systems. In *22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02)*, Vienna, Austria, July 2002. IEEE.

Luc Courtrai, Frédéric Guidec, Nicolas Le Sommer, and Yves Mahéo. Resource management for parallel adaptive components. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS'03)*, page 134b, Nice, France, April 2003.

K. Czarnecki. *Generative Programming : Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, chapter Aspect-Oriented Decomposition and Composition, pages 183–242. 1998.

Pierre-Charles David and Thomas Ledoux. An infrastructure for adaptable middleware. In R. Meersam and Zahir Tari et al, editors, *On the Move to Meaningful Internet Systems 2002 : CoopIS, DOA, ODBASE 2002*, volume 2519 of *Lecture Notes in Computer Science*, pages 773–790. Springer-Verlag, October 2002.

Pierre-Charles David and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In Jean-Bernard Stefani, Isabelle Demeure, and Daniel Hagimont, editors, *Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003*, volume 2893 of *Lecture Notes in Computer Science*, pages 1–14, Paris, November 2003. Federated Conferences, Springer-Verlag.

Linda G. DeMichiel. Enterprise javabeans specification, version 2.1. Sun Microsystems Specification, November 2003.

Anind K. Dey and Gregory D. Abowd. Towards a better understanding of context and context-awareness. In *Workshop on The What, Who, Where, When, and How of Context-Awareness, as part of the 2000 Conference on Human Factors in Computing Systems (CHI 2000)*, The Hague, The Netherlands, April 2000. Also GVU Technical Report GIT-GVU-99-22.

K. R. Dittrich, S. Gatziu, and A. Geppert. The active database management system manifesto : A rulebase of a ADBMS features. In *Proceedings of the 2nd International Workshop on Rules in Database Systems*, volume 985, pages 3–20. Springer-Verlag, 1995.

Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In Don S. Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188, Pittsburgh, PA, USA, October 2002. Springer-Verlag. ISBN 3-540-44284-7.

Jim Dowling and Vinny Cahill. The K-Component architecture meta-model for self-adaptive software. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of Reflection 2001, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan*, volume 2192 of *LNCS*, pages 81–88. AITO, Springer-Verlag, September 2001.

Jim Dowling, Vinny Cahill, and Siobhàn Clarke. Dynamic software evolution and the K-Component model. In *Workshop on Software Evolution, OOPSLA 2001*, 2001.

Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10) :29–32, October 2001.

Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, October 2000. Minneapolis.

Magnus Frodigh, Per Johansson, and Peter Larsson. Wireless ad hoc networking : The art of networking without a network. *Ericsson Review*, (4), 2000.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Professional Computing Series. Addison-Wesley, October 1994.

David Garlan, Bradley Schmerl, and Jichuan Chang. Using gauges for architecture-based monitoring and adaptation. In *Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia, December 2001.

Object Management Group. Corba components, v3.0. OMG Document formal/02-06-65, June 2002.

Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling context information in pervasive computing systems. In Friedemann Mattern and Mahmoud Naghshineh, editors, *First International Conference on Pervasive Computing (Pervasive 2002)*, volume 2414 of *Lecture Notes in Computer Science*, pages 167–180, Zürich, Switzerland, August 2002. Springer-Verlag. ISBN 3-540-44060-7.

John Hughes. Why functional programming matters. *Computer Journal*, 32(2) :98–107, 1989.

Walter Hürsch and Cristina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, Massachusetts, February 1995.

Zahi Jarir. Adaptabilité dynamique des services dans JOnAS. Technical report, École des Mines de Nantes, 2002.

Jeffrey Kephart. A vision of autonomic computing. In Richard P. Gabriel, editor, *Onward ! proceedings from an OOPSLA 2002 track*, pages 13–36, Seattle, WA, USA, November 2002. ACM.

Gregor Kiczales. Beyond the black box : Open implementation. *IEEE Software*, 13(1), January 1996.

Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The art of the Meta-Object Protocol*. MIT Press, 1991.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, 2001.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*. Springer-Verlag, June 1997.

Fabio Kon and Roy H. Campbell. Supporting automatic configuration of component-based distributed systems. In *5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, 1999.

Fabio Kon, Manuel Romàn, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Proceedings of Middleware 2000*, volume 1795 of *LNCS*, pages 121–143. Springer-Verlag, April 2000.

Oussama Layaïda and Daniel Hagimont. Composition et reconfiguration hiérarchiques pour des services multimédia auto-adaptables. In *4ième Conférence Française sur les Systèmes d'Exploitation (CFSE-4)*, Le Croisic, France, April 2005a. Chapitre Français de l'ACM-SIGOPS.

Oussama Layaïda and Daniel Hagimont. PLASMA : A component-based framework for building self-adaptive applications. In *Proceedings of SPIE/IS&T Symposium On Electronic Imaging, Conference on Embedded Multimedia Processing and Communications*, San Jose, USA, January 2005b.

Thomas Ledoux. OpenCorba : a reflective open broker. In *Reflection'99*, volume 1616 of *LNCS*. Springer-Verlag, 1999.

Thomas Ledoux, Mireille Blay, Eric Bruneton, Denis Caromel, Thierry Coupaye, Daniel Hagimont, Jean-Marc Menaud, Jacques Noyé, and Michel Riveill. D1.1 - État de l'art sur l'adaptabilité. Technical Report D1.1, Projet RNTL ARCAD, December 2001.

Baochun Li and Klara Nahrstedt. A control-based middleware framework for quality-of-service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9) :1632–1650, September 1999.

Joseph P. Loyall, Richard E. Schantz, John A. Zinky, and David E. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, Kyoto, Japan, April 1998.

Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, pages 147–155, New York, USA, 1987. ACM SIGPLAN, ACM Press.

Jacques Malenfant, Maria-Teresa Segarra, and Françoise André. Dynamic adaptability : the MolèNE experiment. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of Reflection 2001, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan*, volume 2192 of *LNCS*, pages 110–117. AITO, Springer-Verlag, September 2001.

Masoud Mansouri-Samani and Morris Sloman. Gem : a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2) :96–108, 1997.

Keith Marzullo, Robert Cooper, Mark D. Wood, and Kenneth P. Birman. Tools for distributed application management. *IEEE Computer*, 24(8) :42–51, August 1991.

Jeff McAffer. Meta-level programming with CodA. In Walter Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 190–214. Springer-Verlag, August 1995.

Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi : New-age components for old-fashioned Java. In Linda Northrop, editor, *OOPSLA'01 Conference Proceedings*, pages 211–222, Tampa Bay, Florida, USA, October 2001. ACM, ACM Press.

Douglas McIlroy. *Software Engineering, Report on a conference sponsored by the NATO Science Committee*, chapter Mass-Produced Software Components, pages 138–155. Scientific Affairs Division, NATO, Garmisch, Germany, October 1968.

Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), January 2000.

Kaveh Moazami-Goudarzi. *Constistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College, London, March 1999.

Douglas Moreto and Markus Endler. Evaluating composite events using shared trees. Rapport Technique RT-MAC-2001-01, IME-USP, January 2001.

Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 276–287, Saint Malo, France, 1997. ACM Press.

Object Management Group. Common object request broker architecture (CORBA/IIOP), version 2.5. OMG Document formal/2001-09-01, September 2001.

Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3) :54–62, May 1999.

Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98)*, Kyoto, Japan, April 1998.

Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Research Report RC 21451(96717)16APR99, IBM T.J. Watson Research Center, April 1999.

Klaus Ostermann, Mira Mezini, and Christoph Bockish. Expressive pointcuts for increased modularity. In *Proceedings of ECOOP 2005*, Lecture Notes in Computer Science, Glasgow, UK, July 2005. Springer-Verlag.

Partha Pal, Joseph Loyall, Richard Schantz, John Zinky, Rich Shapiro, and James Megquier. Using QDL to specify QoS aware distributed (QuO) application configuration. In *Proceedings of ISORC 2000, The 3rd IEEE International Symposium on Object-Oriented Real-time distributed Computing*, Newport Beach, CA, March 2000.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12) :1053–1058, December 1972.

Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC : A flexible and efficient solution for aspect-oriented programming in Java. In *Reflection 2001*, volume 2192 of *LNCS*, pages 1–24. Springer-Verlag, September 2001.

Nicolas Pessemier, Lionel Seinturier, Laurence Duchien, and Olivier Barais. Une extension de fractal pour l'AOP. In *Première Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA'04)*, Paris, September 2004.

Viven Quema and Luc Bellissard. Configuration de middelware dirigée par les applications. In *Actes des Journées Composants 2002 – Systèmes à Composants Adaptables et Extensibles*, Grenoble, France, October 2002.

Pierre-Guillaume Raverdy and Rodger Lea. Reflection support for adaptive distributed applications. In *Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC '99)*, 1999.

Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of ECOOP 2002*, volume 2374 of *Lecture Notes in Computer Science*, pages 205–230, Malaga, Spain, May 2002. Springer-Verlag.

Manuel Romàn, Fabion Kon, and Roy H. Campbell. Reflective middleware : From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), 2001. URL `http://www.computer.org/dsonline/0105/features/rom0105_print.htm`.

Andreas Schade. An event framework for CORBA-based monitoring and management systems. In *Joint International Conference on Open Distributed Processing (ICOPD) and Distributed Platforms (ICDP)*, Toronto, Canada, May 1997.

Beth A. Schroeder. On-line monitoring : a tutorial. *IEEE Computer*, 28(6) :72–78, June 1995. URL `http://csdl.computer.org/comp/mags/co/1995/06/r6072abs.htm`.

Murray Shanahan. The event calculus explained. In M. J. Wooldridge and M. Veloso, editors, *Artificial Intelligence Today*, volume 1600 of *Lecture Notes in Artificial Intelligence*, pages 409–430. Springer-Verlag, 1999.

Brian Cantwell Smith. Reflection and semantics in Lisp. In *11th annual ACM Symposium on Principles of programming languages*, pages 23–35, Salt Lake City, Utah, USA, 1984.

Richard Staehli, Frank Eliassen, and Sten Amundsen. Designing adaptive middleware for reuse. In *Middleware 2004 Companion, 3nd Workshop on Reflective and Adaptive Middleware*, 2004.

Jean-Bernard Stefani. A calculus of kells. In *Proceedings of the 2nd International Workshop on Foundations of Global Computing*, Eindhoven, the Netherlands, June 2003.

Clements Szyperski. *Component Software*. ACM Press, New York, 1997.

Éric Tanter, Jacques Noyé, and Denis Caromel an Pierre Cointe. Partial behavioral reflection : Spatial and temporal selection of reification. In Ron Crocker and Jr. Guy L. Steele, editors, *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA 2003)*, pages 27–46, Anaheim, California, USA, October 2003. ACM Press.

Peri Tarr and Lori A. Clarke. Consistency management for complex applications. Technical Report 97-46, Computer Science Department, University of Massachusetts at Amherst, September 1997.

Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation : Multi-dimensional separation of concerns. In *ICSE'99*, Los Angelese, USA, 1999. ACM, ACM Press.

Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages : An annotated bibliography. *ACM SIGPLAN Notices*, 35(6) :26–36, June 2000.

Nanbor Wang, Douglas C. Schmidt, , and Carlos O'Ryan. *Component-Based Software Engineering*, chapter An Overview of the CORBA Component Model. Addison-Wesley, Reading, Massachusetts, 2000.

Roy Want, Andy Hopper, Veronica Falcão, and Jonathan Gibbons. The active badge location system. *ACM Transactions on Information Systems*, pages 91–102, 1992. ISSN 1046-8188.

Michel Wermelinger and José Luiz Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2) :133–155, August 2002.

World Wide Web Consortium. XML path language (xpath) version 1.0. W3C Recommendation, November 1999. `http ://www.w3.org/TR/xpath`.

John Zinky, Joseph Loyall, and Richard Shapiro. Runtime performance modeling and measurement of adaptive distributed object applications. In R. Meersam and Zahir Tari et al, editors, *On the Move to Meaningful Internet Systems 2002 : CoopIS, DOA, ODBASE 2002*, volume 2519 of *Lecture Notes in Computer Science*, pages 773–790, Irvine, California, USA, October 2002. Springer-Verlag.

# Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation

Pierre-Charles David

**Résumé**

Les contextes toujours plus variés et dynamiques dans lesquels les logiciels actuels s'exécutent leurs imposent de s'adapter de façon *autonome* à ces changements. L'objectif de cette thèse est de faciliter le développement de telles *applications adaptatives*, en considérant l'adaptation comme un *aspect* qui doit être développé séparément du reste de l'application afin de pouvoir y être intégré et modifié dynamiquement. Pour cela nous proposons SAFRAN, une extension du modèle de composants Fractal permettant d'associer dynamiquement des politiques d'adaptation aux composants d'une application. Ces politiques sont programmées dans un langage dédié sous la forme de règles réactives. Leur exécution repose d'une part sur WildCAT, un système permettant de détecter les évolutions du contexte d'exécution (quand adapter ?), et d'autre part sur FScript, un langage dédié pour la reconfiguration dynamique consistante de composants Fractal (comment adapter ?).

**Mots-clés :** logiciels adaptatifs, séparation des préoccupations, politiques d'adaptation, langages dédiés, règles réactives, reconfiguration dynamique, sensibilité au contexte, composants logiciels, Fractal

**Abstract**

The increasingly diverse and dynamic contexts in which current applications are run imposes them to adapt and to become more *autonomous*. The goal of this thesis is to ease the development of such *self-adaptive applications*, by considering adaptation as an *aspect* which should be defined separately from the rest of the application, so as to be integrated and modified dynamically. To this end we propose SAFRAN, an extension of the Fractal component model enabling dynamic association of adaptation policies to the components of an application. These policies are programed using a Domain-Specific Language in the form of reactive rules. Their execution harnesses WildCAT, a context-awareness system which can detect changes in the execution context (when to adapt?), and FScript, a language dedicated to dynamic and consistent reconfigurations of Fractal components (how to adapt?).

**Keywords:** self-adaptive software, separation of concerns, adaptation policies, domain-specific languages, reactive rules, dynamic reconfiguration, context-awareness, software components, Fractal